PKUAS 2010 技术手册

北京大学

PKUAS 2010 技术手册

北京大学

目录

插图清单

前言

1. 关于JEE应用服务器

应用服务器是一种为方便应用软件开发与维护而逐步形成的基础软件，它是网络环境中应用系统的高层运行平台，其主要目的是使应用系统的代码更为简单，使开发人员的精力可以更加集中于系统的逻辑部分。

JEE应用服务器是指符合J2EE or Java Enterprise Edition 5规范的应用服务器。

2. 关于PKUAS

构件运行支撑平台PKUAS是由北京大学信息科学技术学院软件研究所自行设计开发，兼容Java EE 5(Java Platform, Enterprise Edition 5)规范、面向Internet的 开放式构件运行支撑平台。PKUAS在具有全面功能的同时，易于配置、管理，对资源要求低而运行效率高，具有出色的性价比特征，非常适合中小企业、政府部门等。PKUAS具有开放的构件化体系结构，具有强大的互操作能力和出色的定制与扩展能力，可以为企业量体裁衣，方便灵活地构造面向特定领域的应用中间件。同时，PKUAS最大限度为用户着想，能够在不中断系统运行的情况下对应用系统进行在线演化，保护关键业务的进行，具备下一代中间件的领先特征。

PKUAS兼容Java EE 5规范和Enterprise JavaBean3.0规范中的核心内容， 全面支持基于J2EE/EJB体系结构的应用， 已经通过了J2EE蓝图程序Java PetStore（JPS） （1.3版本） 、J2EE性能基准测试程序ECperf(1.0版本)和ObjectWeb发布的开源Jonas兼容性测试集的测试。

PKUAS占用资源少，对系统硬件配置要求低，可在奔腾166MHz、48兆内存、200M空闲磁盘空间的机器上正常工作。PKUAS支持多种操作系统：Windows 2000, Windows XP，Windows Server 2003, Windows Vista, Win 7和Unix/Linux系统。PKUAS要求Java SE 1.6或以上版本支持。

3. 关于本书

本书面向PKUAS的用户，也可以供PKUAS应用服务器开发人员参考。

本书介绍了PKUAS的设计和实现，目的是便于用户了解和使用PKUAS应用服务器。书中对所涉及到的JEE相关内容不作具体介绍，这方面内容请参看SUN公司有关文档。

本书一共分为十六章。第一章引言，第二章介绍PKUAS的安装与配置，第三至十四章分别从体系结构、容器与服务等方面介绍PKUAS的设计和实现，第十五章是基于PKUAS的应用系统开发指南，第十六章列出了本书的参考资料。

第 1 章 安装与配置

1.1. 第1小节标题

1.1.1. 第1.1小节标题

第1.1小节内容

1.1.2. 第1.2小节标题

第1.2小节内容

1.2. 第2小节标题

1.2.1. 第2.1小节标题

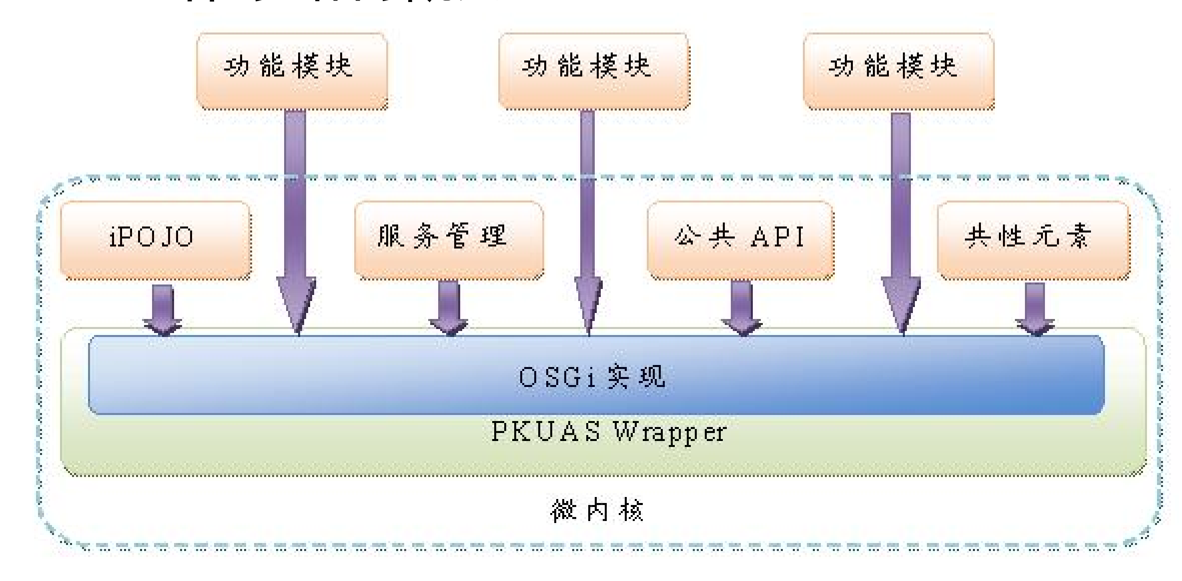
第2.1小节内容

1.2.2. 第2.2小节标题

第2.2小节内容

第 2 章 体系架构

2.1. PKUAS体系结构概述



PKUAS的体系结构如上图所示，采用“微内核+功能模块”的结构。下面分别介绍：

- 微内核：PKUAS的微内核采用符合OSGi Release 4规范的内核实现Felix。运行在微内核上的每一个构件模块称为一个Bundle。微内核负责管理Bundle的生命周期，包括运行时刻的动态安装、卸载、启动和停止等。Felix实现之上还包含以下几个属于微内核的模块（以下模块同时也作为Bundle部署在OSGi平台上）：

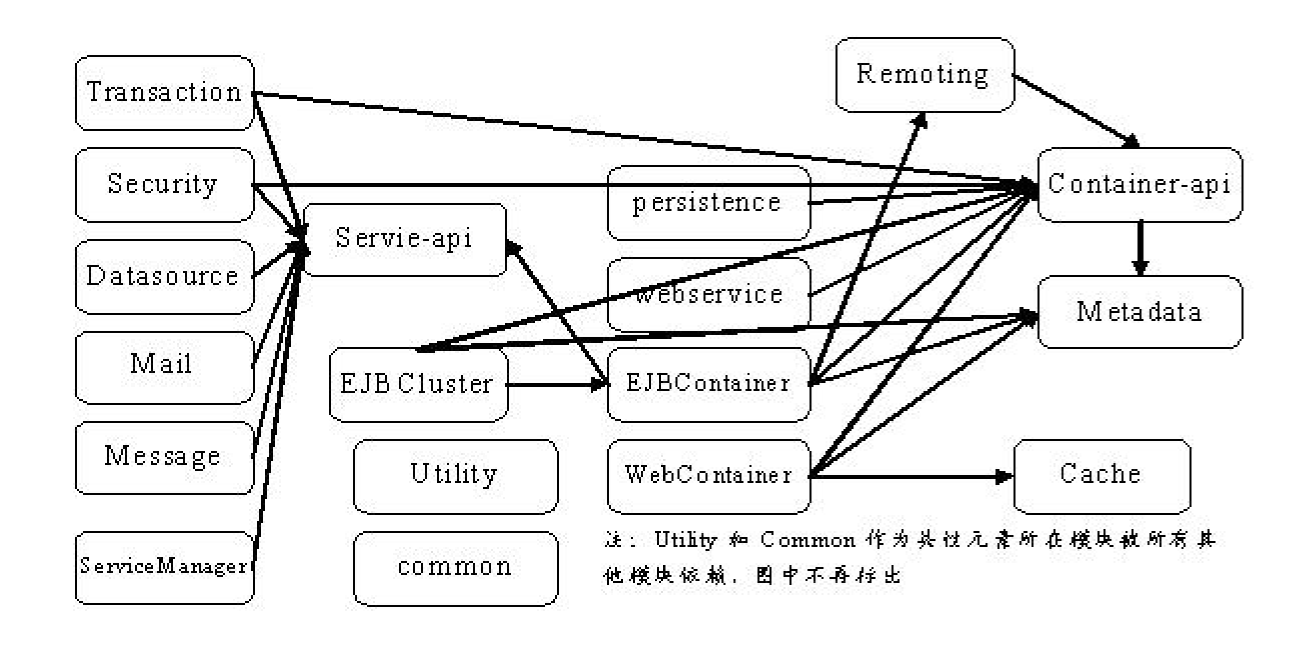
  1.iPOJO：iPOJO是一个基于OSGi的服务构件模型，其主要作用是简化OSGi的Bundle开发。它仅仅通过一个metadata.xml描述文件，就可以实现Bundle的生命周期和动态依赖管理。

  2.服务管理：该模块完成被管理对象类的管理和配置功能。其中，被管理对象类是指每个功能模块中实现了被管理接口、可以被微内核管理生命周期的类。服务管理与iPOJO交互实现管理和配置功能。

  3.公共API：这个模块实际上是PKUAS的SPI(Service Provider Interface)。它抽取了所有模块接口类的interface，并集合在一起形成了公共API模块。这样，模块之间通过依赖注入实现了松散耦合。

  4.共性元素：主要包括日志、对象序列化等所有模块的AOP(Aspect Oriented Programming)元素，PKUAS公共类及被多个模块使用的第三方包。

- 功能模块：每一个PKUAS功能模块作为一个OSGi Bundle，实现不同的功能。PKUAS划分的模块及模块之间的相互关系如下图所示：



其中的功能模块主要分为三类：

1.容器系统：容器是构建运行时所处的空间，负责构件的生命周期管理以及构件运行需要的上下文管理，同时为系统的非功能性约束（例如安全、事务等）提供服务接入点。其中抽取的接口形成Container-api模块。

2.服务：实现系统中与业务无关的功能和约束，如通信、安全、事务等。PKUAS的服务可以通过微内核动态的增加、替换和删除，具有很强的可定制性和灵活性。具体说来，PKUAS采用了一种松耦合的服务集成框架，通过引入适配器模型和反射机制，容器系统的实现仅仅依赖于服务抽象出来的Service-api，而用ServiceInfo文件描述具体装载哪个服务实现，从而实现容器系统和服务实现的演化相互独立。

3.共性元素：提供被多个模块依赖的功能。

2.2. 应用服务器启动过程

1.pku.as.launcher.felix.PKUAS从配置文件default-config.properties和pkuas-autodeploy-bundles.properties获取将要启动的Bundle列表（其中deafault-config.properties声明Felix所需要的系统Bundle，pkuas-autodeploy-bundles.properties声明PKUAS默认启动的Bundle），然后启动Felix内核。

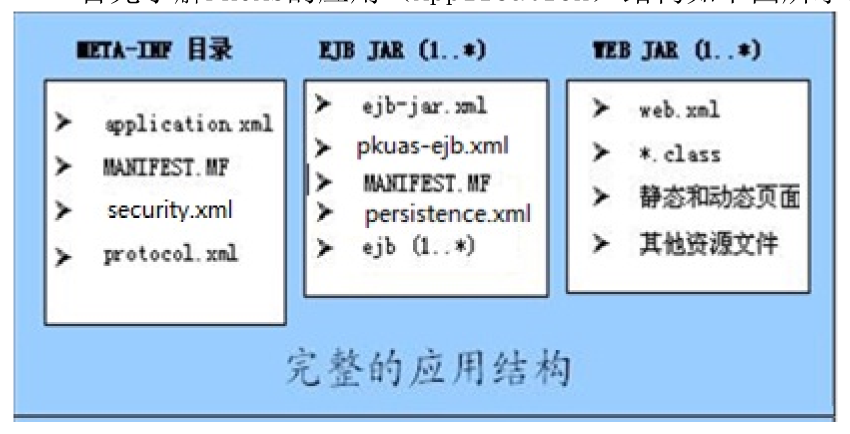
2.Felix内核根据pkuas-autodeploy.properties中声明的顺序加载各个Bundle。

3.Felix内核创建Service Manager的一个实例，由Service Manager完成Bundle Service的启动。

4.Bundle启动完毕之后控制流转到Application Manager，由它加载所有的应用。

2.3. 应用部署和启动过程

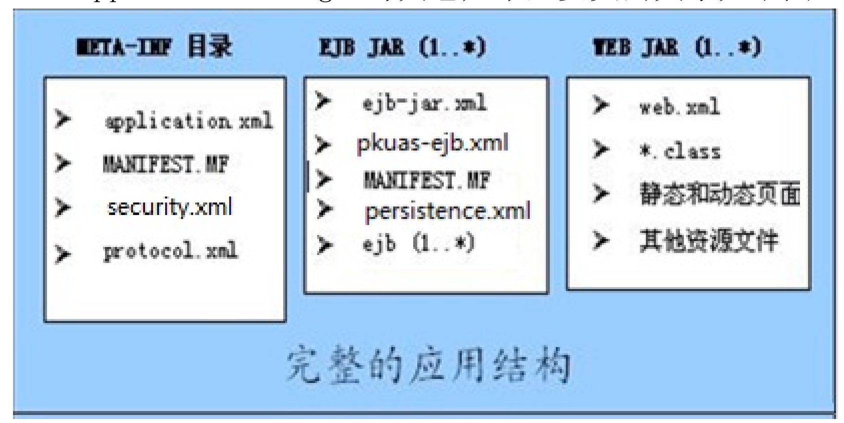
首先了解PKUAS的应用（Application）结构如下图所示：



PKUAS的部署单位为EJB应用或者Web应用。也就是说，不能直接部署整个应用（包括EJB和Web构件的EAR文件），必须分别部署EJB构件和Web构件。

ApplicationManager（pku.as.application）类负责对应用进行管理、部署、卸载、重新部署、启动和停止，该类使用了Singleton设计模式，以保证在一个PKUAS中只有一个ApplicationManager实例。

ApplicationManager与其它几个重要类的关系如下图：



2.4. 应用部署的主要步骤

在Application中实现，有两大步骤:

1.生成application的信息。包括应用本身的信息以及应用部署的信息。

2.对application进行部署。包括生成application对应的容器，生成通讯器，生成客户端容器，装配intercept链，绑定JNDI等。

2.5. 应用部署的主要流程

在ApplicationManager.loadApplication中实现，主要包括两个步骤：

1.把所有jar包和ear包拷到deployed目录下，然后对每一个调用deployApp方法进行部署。

2.ApplicationManager.deployApp部署一个应用:

(1).调用ApplicationManager.construcApp方法来构建一个application，得到这个应用的实例（具体过程在下面介绍）。

(2).为application设置包括命名、事务、安全在内的方法引用。

(3).调用application.deploy方法对应用进行部署。

(4).生成一个线程AutoDeployThread来监控部署的应用的变化。

2.6. 如何构建一个应用

ApplicationManager.construcApp方法

1.目的：主要是为了生成application的信息ApplicationInfo，包括它本身内部EJB和Web应用的信息以及部署信息。

2.构建的流程

(1).判断该应用是否更新过（通过AppInfoStore类的最近更新时间）。

(2).如果更新过：调用deployCMP方法新进行解压缩。先进行CMP转换处理，对于CMP，要调用CMP20Engine.work方法将CMP转换为BMP。然后拷贝并解开jar包或ear包。

(3).生成AppClassLoader，地址设为部署目录和服务器端的codebase。

(4).生成ApplicationDeployer，调用它的buildApplication方法，目的是解析各种配置文件和EJB、Web应用信息，最终得到应用的信息ApplicationInfo（具体过程在下面介绍）。

(5).通过ApplicationFactory得到Application实例。

(6).生成application的信息ApplicationInfo。

3.由ApplicationDeployer实现application的ApplicationInfo的生成。ApplicationDeployer的buildApplication方法得到application的各种配置文件中已经的部署信息。

(1).Xml文件用castor解析:通过三个Parse过程（parserApplicationXML，parserSecurityXML，parserProtocolXML）解析application.xml,security.xml,protocol.xml，得到application内部的各个module。

(2).解析EJB的jar包（processEJB）：为内部每个bean装配EJBClassLoader，将每个类存到classList中。

(3).解析Web的war包（processWeb）。

(4).构建EJB（buildEJBcomponents）。

• 解析pkuas-ejb.xml。

• 解析ejb-jar.xml（ejb部署信息，解析注解时优先级比代码的高）：解析每个Bean，得到元数据信息bmi（支持EJB2.0和3.0）

• 解析式通过调用ParserManager的processAnnotaion方法，返回一个class对应的bmi。

• ParserManager中提供MDB，SLB，SFB三种parser，分别调用它们的processAnnotation方法进行解析。

• 过程遵循EJB规范。

• 根据上面解析得到bmi的类型生成具体的BeanDeploymentInfo（EJB2.0中是SessionMetaData，EntityMetaData和 MessageDrivenMetaData，3.0中只有SessionMetaData ， MessageDrivenMetaData），作为部署的具体信息。

• 构建Web构件（buildWebComponents。

(5)初始化每个构件的bdi，做本地JNDI名绑定（只在EJB3.0中实现）。

4.构建EJB的buildEJBcomponents方法中，提供了加入一组InjectionDeployer的步骤，表示用户可以加入自己的deployer，实现对特定EJB定制自己的部署方式，这个deployer和ApplicationDeployer在地位上是平行的，都会被执行，在ServerConfig的getDeployerList方法中来获得deployer，所以用户要部署什么样的deployer，可以在ServerConfig的这一方法中获得，通常应该是读取一个关于部署器的配置文件，此处没有实现这一功能。

2.7. 对application进行部署

1.在ApplicationManager中生成application后，调用application的deploy方法进行部署。

2.采用动态代理调用deploy。

(1).目的：为了能将applicaton注册到MBeanServer中进行管理。

(2).自动生成动态代理ApplicationProxy，将deploy和对MBeanServer的注册（在ApplicationManagermentInterceptor类的intercept方法中进行）都放在它的deploy方法中执行。

3.在deploy方法中调用start方法启动应用，包括启动JRMP通讯器服务，生成通讯器的stub，生成客户端容器代理类ClientContainer。

4.生成容器实例：用哈希表contains存储应用中用到的所有容器，messageDrivenContainers存储用到的消息Bean的容器，取出应用中所有的构件，分别生成对应的容器，将容器注册到MBeanServer中，并进行初始化（主要是

生成interceptor链）消息Bean容器和其他容器的处理方法不一样，消息Bean容器时直接new出实例，然后进行注册，其他容器是用EJBContainerFactory生成，同时为容器设置事务、安全、命名等服务；启动Web应用。

5.完成部署后，启动一个AutoDeployThread线程，用于监听部署的应用在运行时的变化。

第 3 章 EJB 容器

3.1. 概述

应用服务器中容器的作用主要是：

1.容器是异构的EJB构件的运行支撑环境。实现容器与构件的合约；维护构件运行的上下文（如安全与事务上下文）；管理构件的生命周期（如类装载、实例化、缓存、释放等），主要由类装载器、实例池与相应的对象持久化机制实现。

2.容器是为分布的EJB构件提供远程访问能力的载体。EJB调用者使用命名查找的方式得到EJB的远端代理，用来和服务器端的EJB进行交互。容器需要为分布式环境下的EJB生成相应的远端代理。

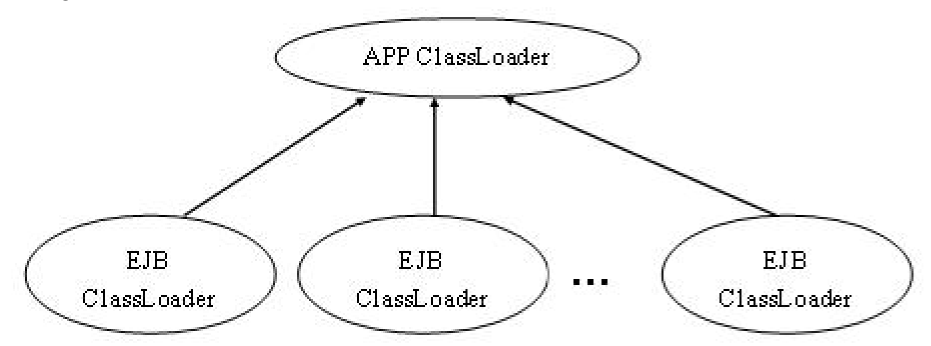
3.构件容器为受控的EJB构件提供企业计算特性。容器需要为系统的非功能约束（安全、事务等）提供检查点以实现关注点分离，供公共服务接入。

在PKUAS的容器系统中，每一类EJB对应一种容器，包括无态会话构件容器、有态会话构件容器、实体构件容器、以及消息驱动的构件容器。应用（Application）将相关联的容器组织到一起。

PKUAS中的应用（Application）是容器系统管理构件的基本组织单位，与实际部署的应用程序包一一对应的。一个构件种类对应一个容器类型，而应用程序包里的一个构件对应一个容器实例。应用实际上是容器管理框架的主要实现。首先，它是对容器的一种分组，将逻辑上相关的容器实例组织在一起，防止不同应用之间构件的相互干扰。同时，每一个应用都有自己独立的类加载器，可以防止非本应用的代码随意访问应用中的构件，从而提高了安全性。其次应用将容器、通讯器和服务集成在一起，构成了构件平台的核心。应用在启动时，根据应用程序包中的部署文件，装载相应的通讯器、容器以及容器中的服务截取器。在PKUAS中，每个应用对应一个通讯器的，由应用负责接收客户的请求，然后转发给相应的容器处理。

3.2. 类加载机制

PKUAS中每一个应用采用两层ClassLoader结构，第一层是应用的类加载器AppClassLoader（pku.as.container.app，见modules/apis/container-api/src/main/java），第二层是各种类型的EJB ClassLoader。如下图所示:

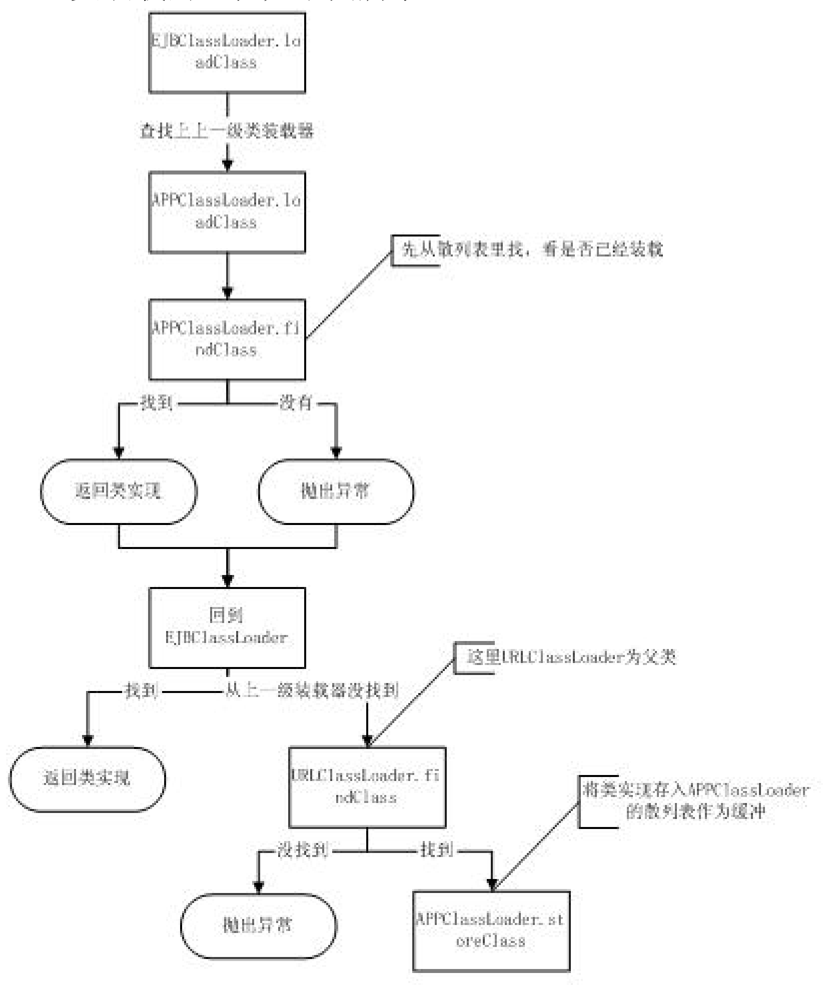


每一个PKUAS的应用都对应一个自己的AppClassLoader实例。AppClassLoader继承URLClassLoader（java.net.URLClassLoader，用于从指向 JAR 文件和目录的

URL 的搜索路径加载类和资源），用一个散列表来存储已经加载的类的类名和实现之间对应关系，并提供方法装载某个构件中所有的类。

EJBClassLoader（pku.as.container.ejb.EJBClassLoader）类同样继承URLClassLoader，并包含一个指向父AppClassLoader的引用，它进行具体的类装载。

类装载的流程如下图所示：



EJBClassLoader在装载一个类时，如果找不到该类，首先委托上一层的AppClassLoader装载。AppClassLoader则在已装载类的散列表中寻找该类，如果找到，直接返回类实现，否则，再委托URLClassLoader装载类，并将类实现加入在散列表中。

3.3. 客户调用工作流程

对于一个调用，PKUAS的工作流程如下：

1.客户调用由应用服务器提供的命名服务查找构件的类型接口。

2.客户调用类型接口的创建方法，构件stub（运行时刻由Interceptor动态生成）将客户请求发送给服务器。

3.服务器传输层接收到客户请求后，根据请求中的标识，分发给相应的容器处理，即调用相应容器的invoke（pku.as.ejb.Container）方法。

4.Invoke方法调用容器初始化时注册在容器中的Interceptor的intercept（pku.as.DefaultInterceptor）方法，该方法负责依次调用Interceptor的beforeInvoke方法进行调用前的处理。

5.容器调用实例管理器获得一个实例，并生成相应的构件标识（Instance ID）。

6.容器依次调用各个截取器的afterInvoke()接口（步骤4和6中的beforeInvoke和afterInvoke方法是进行客户端的处理），进行调用后的处理。

7.容器返回构件标识（Instance ID）。

8.通讯机制将构件标识包装成构件的实例引用。（这个应该由通讯框架完成）

9.客户得到实例引用后，调用其中的实例方法，构件stub将请求发送给服务器。

10.服务器接收到客户请求（InvocationRequest）后，根据请求中的标识（方法名），分发给相应的容器处理。

11.容器依次调用各个截取器的beforeInvoke()接口，进行调用前的处理。

12.容器调用实例管理器，根据请求中包含的构件标识，调用命名服务的lookup方法得到指定的构件实例。

13.容器调用构件实例处理用户请求，通过反射机制invoke用户指定的方法。

14.容器依次调用各个截取器的afterInvoke()接口（步骤11和步骤14中的beforeInvoke和afterInvoke方法是进行服务器端的处理），进行调用后的处理。

15.容器返回调用结果。

16.客户得到调用结果。

3.4. 实例管理

EJB容器用实例池化（instance pooling）技术来管理大量Bean的实例。因为EJB的客户端从不直接访问Bean的实例，而是通过容器来访问，因此，容器只需要维护少量的Bean实例，并重复使用这些Bean实例为不同的请求提供服务就可以了。这种技术可以大大减少处理全部请求所需要耗费的资源总量。实例管理的基类是pku.as.ejb.im.BaseInstanceManager。不同类型的EJB实例管理类继承该类。

1.无态会话Bean的实例管理（pku.as.ejb.stateless.StatelessInstanceManager）：无态会话Bean不需要维护任何内部状态，每次独立的方法调用也不依赖于任何实例变量，因此它的实现相对简单。一个无态会话Bean的生命周期存在三种状态：

- 无状态：Bean实例尚未被初始化的状态，通常表示一个Bean实例生命周期的开始和结束。

- 池状态：Bean实例已经被容器初始化，但是尚未与一个EJB请求相关联。
- 就绪状态（ReadyKey）：Bean实例已经与一个EJB请求相关联，并且已经做好了响应业务方法调用的准备。

2.消息驱动Bean的实例管理（pku.as.ejb.message.MessageDrivenInstanceManager）：同无状态会话Bean一样，消息驱动Bean也不需要维护任何内部状态。因此当消息到来时，实例管理器只要简单的调用fetchFromPool方法从实例池中取出一个实例（若实例池中没有可用的实例，则重新new一个实例），使用完毕后调用toPool方法把实例放回实例池中即可。

3.有态会话Bean的实例管理（pku.as.ejb.stateful.StatefulInstanceManager）：有态会话Bean的实例管理最大的区别是需要在不同的方法调用之间维护会话状态，实例管理器要负责对实例进行钝化（passivation，解除有态会话bean实例与相关EJB对象之间的关联，并保存其状态的操作）和激活（activate，根据所关联的EJB对象将Bean实例的状态重新回复的操作）操作。因此实例管理器需要保存实例钝化器（pku.as.ejb.stateful.StatefulPassivator）的引用。

4.实体Bean的实例管理（pku.as.ejb.entity.EntityInstanceManager）：实例管理器负责实现实例的钝化和激活操作。

在PKUAS中，实例池的实现采用两种数据结构，一种是集合，一种是队列。

1.队列：处于队列中的构件实例没有标识，处于池状态的Bean。

2.散列表：处于散列表中的每一个构件实例都有惟一标识，用于处于就绪状态的Bean（包括处于会话中的有态会话Bean）。

此外，实例池在实现中把处于事务中的和没有处于事务中的实例分开管理。

第　4　章　通信框架

4.1.　第1小节标题

4.1.1.　第1.1小节标题

第1.1小节内容

4.1.2.　第1.2小节标题

第1.2小节内容

4.2.　第2小节标题

4.2.1.　第2.1小节标题

第2.1小节内容

4.2.2.　第2.2小节标题

第2.2小节内容

第 5 章 管理框架

5.1. 第1小节标题

5.1.1. 第1.1小节标题

第1.1小节内容

5.1.2. 第1.2小节标题

第1.2小节内容

5.2. 第2小节标题

5.2.1. 第2.1小节标题

第2.1小节内容

5.2.2. 第2.2小节标题

第2.2小节内容

第　6　章　WEB容器

6.1.　第1小节标题

6.1.1.　第1.1小节标题

第1.1小节内容

6.1.2.　第1.2小节标题

第1.2小节内容

6.2.　第2小节标题

6.2.1.　第2.1小节标题

第2.1小节内容

6.2.2.　第2.2小节标题

第2.2小节内容

第 7 章 命名服务

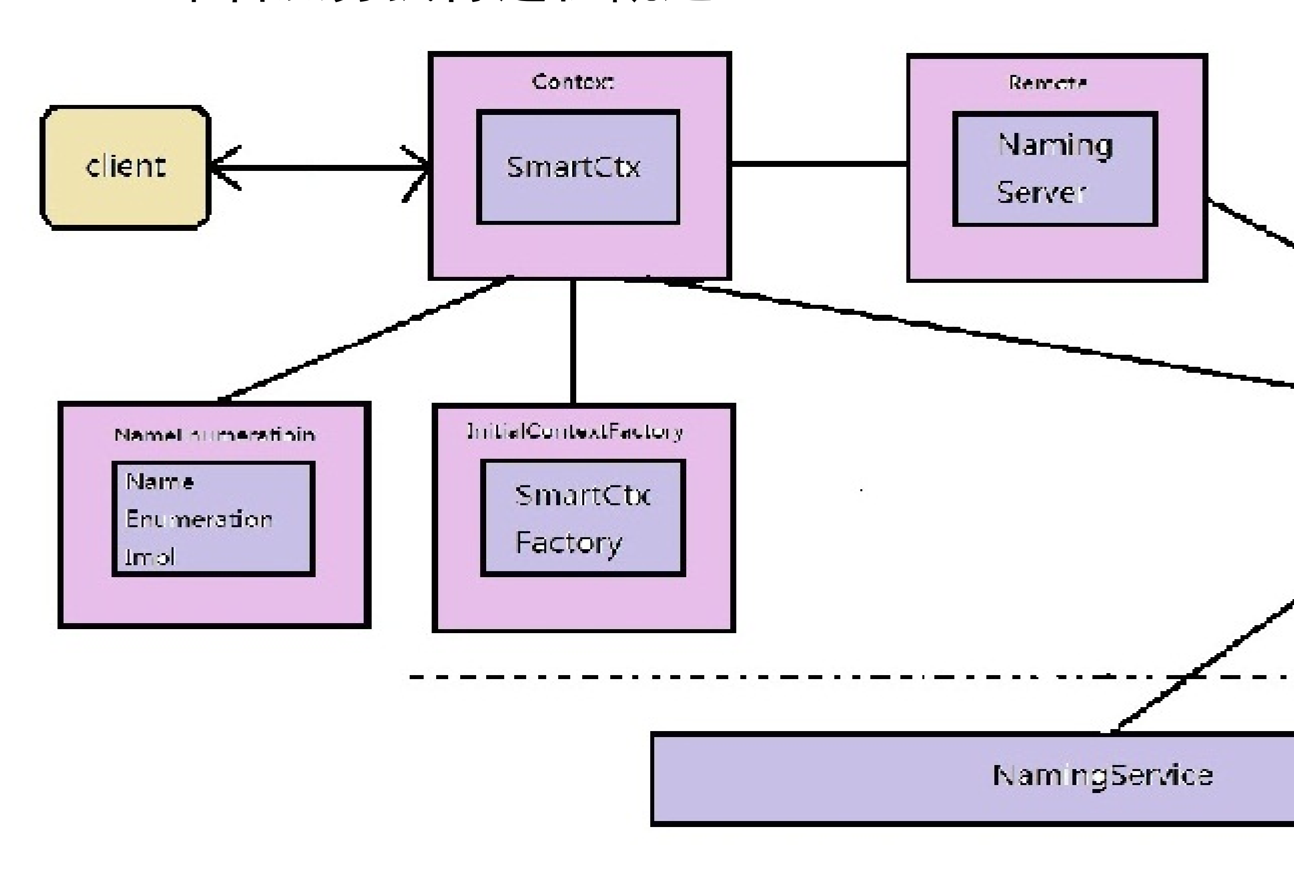
7.1. PKUAS命名服务机制概述

命名服务(Naming Service)提供了一种为对象命名的机制，使得用户可以在无需知道对象位置的情况下获取和使用对象。只要该对象在命名服务器上注册过，并且用户知道命名服务器的地址和该对象在命名服务器上注册的JNDI名，就可以找到该对象，获得其引用，从而使用它提供的服务。

在目前的PKUAS实现中，用户如果要使用命名服务，首先通过一个用户调用接口SmartCtx来查找需要的对象，PKUAS会把SmartCtx的请求转发给相应互操作协议的命名服务的代理，代理再把请求转发到具体的命名服务实现类。

目前PKUAS的命名服务实现类为NamingServer类，它维护一个从名字到对象的ConcurrentHashMap，应用在部署的时候将自身所包含的EJB名字和对象的映射关系插入到散列表中，查找对象时只需要在散列表中寻找即可。

7.2. PKUAS命名服务执行过程概述

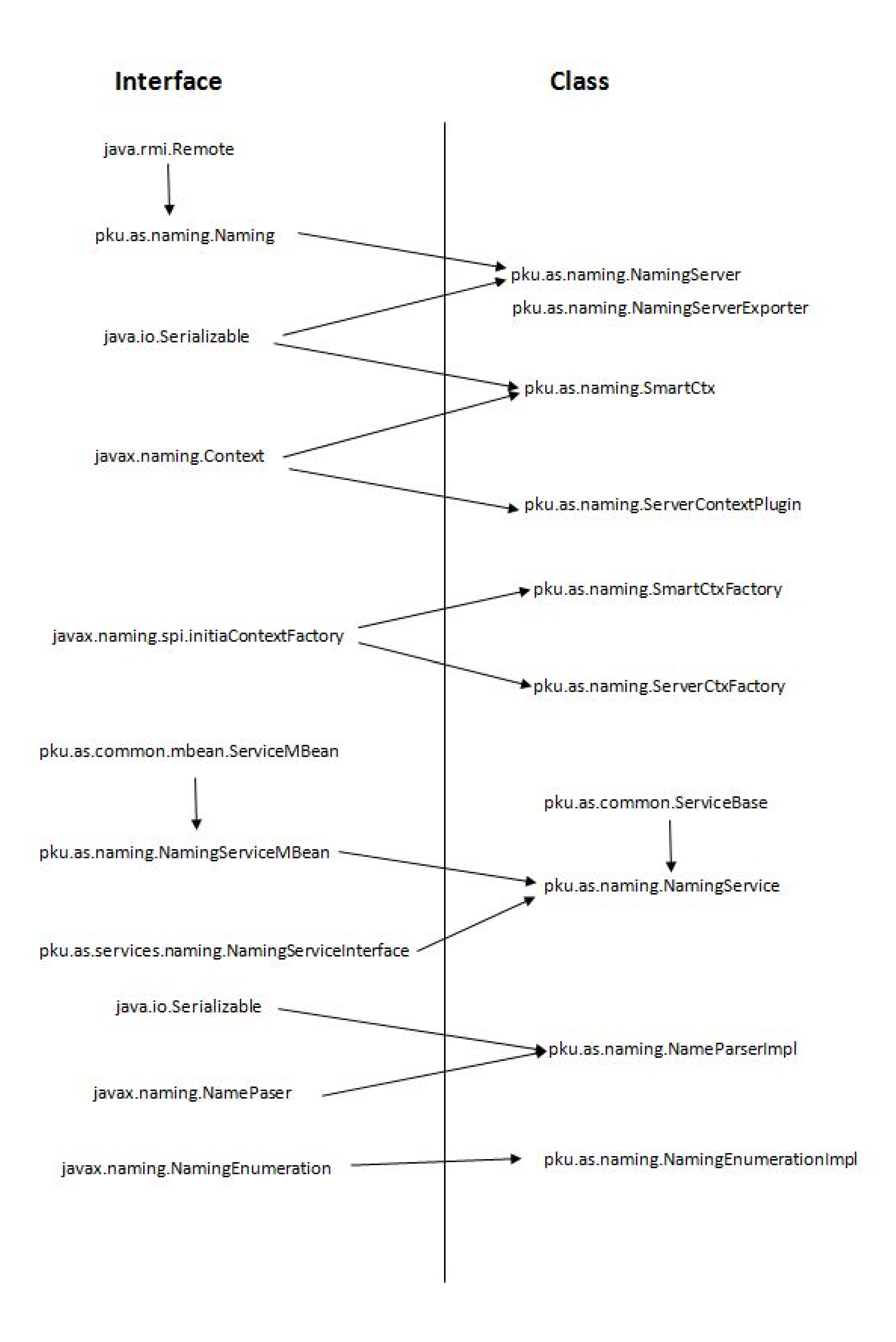


- 客户端发出SmartCtx上下文请求
- SmartCtxFactory负责生成具有一定环境属性的SmartCtx初始上下文

- SmartCtx调用NamingServer中的方法执行客户端请求并得到一个结果
- 对于返回对象集的操作，还要调用NameEnumerationImpl中的方法枚举所有对象
- 整个命名服务由NamingService管理
- 在SmartCtxFactory、NamingServer和NamingService中处理名称时使用NameParserImpl进行名称解析

7.3. PKUAS命名服务包——pku.as.naming

7.3.1. 类和接口概览



7.3.2. 接口概述

7.3.2.1. pku.as.naming.Naming

pkuas命名服务对外提供的接口。

该接口扩展了javax.rmi.Remote接口，声明了一组上下文操作（建立/重新/消除绑定、查找、两种列表、创建/销毁子上下文），是远程虚拟机可调用的本地虚拟机方法。

实现：

pku.as.naming.NamingServer

7.3.2.2. pku.as.naming.NamingServiceMBean

pkuas中，命名服务作为消息Bean被管理。这个接口就是对外提供了将命名服务作为消息Bean的管理方法。它扩展了pku.as.common.mbean.ServiceMBean接口，提供获取和设置数据端口和命名端口的抽象方法。

实现：

pku.as.naming.NamingService

7.3.3. 类概述

7.3.3.1. pku.as.naming.NamingServer

命名服务器类是pkuas命名服务的核心。

此类实现了pku.as.naming.Naming和 java.io.Serializable两个接口，可序列化地实现了pku.as.naming.Naming中声明的命名上下文操作（建立/重新/消除绑定、查找、三种列表、创建/销毁子上下文），供SmartCtx中的方法调用。

NamingServer是所有命名服务的实际执行者。

在NamingServer中，首先以map为基础，实现一组基本操作，再调用这组基本操作来完成Naming中声明的上下文操作。

如果名称是一个CompoundName，在绑定第k个component时，要求前k-1个component都已经绑定到了对象，并且要求其第一个component绑定在一个命名服务器上。

对于返回子上下文的列表方法，NamingServer返回给SmartCtx一个Collection对象，再由SmartCtx将Collection对象转化为NameEnumeration对象返回给客户端。

7.3.3.2. pku.as.naming.SmartCtx

SmartCtx是pkuas命名服务器和客户端进行交互的上下文类，即客户端发出SmartCtx的请求，pkuas返回SmartCtx的响应。

此类实现了javax.naming.Context和java.io.Serializable两个接口。

提供resolveObject()方法，对对象进行预处理，将非MarshalledObject变成MarshalledObject，便于将对象传入rmi调用；

提供resolveName()方法，对名称进行预处理，对于以"java:"开头的名称，解析它之后的部分；对于以空开头的名称，解析它之后的名称，对于其它非空名称，给它增加前缀。

调用命名接口，实现命名上下文的所有操作（建立/重新/消除绑定、查找、两种列表、创建/销毁子上下文），而不同于命名服务器，它对每种名称为Name类型参数的方法，都名称为String类型参数的相应方法：对于名称为Name类型的操作，首先调用resolveName()方法和resolveObject方法对名称和对象进行预处理；对于名称为String类型的操作，直接用命名解析器解析这个名称。

7.3.3.3. pku.as.naming.ServerContextPlugin

服务器上下文插件，实现了javax.naming.Context接口。

7.3.3.4. pku.as.naming.SmartCtxFactory

SmartCtx工厂，实现了InitialContextFactory接口，用于创建具有特定环境属性值的SmartCtx初始上下文。

7.3.3.5. pku.as.naming.ServerCtxFactory

ServerCtx工厂，实现了InitialContextFactory接口，用于创建具有特定环境属性值的ServerCtxPlugin初始上下文。

7.3.3.6. pku.as.naming.NamingEnumerationImpl

命名枚举实现，实现了javax.naming.NamingEnumeration接口。主要是将NamingServer返回的非序列化的对象集合转化为序列化的对象枚举，返回给客户端。

7.3.3.7. pku.as.naming.NamingService

该类实现了NamingServiceMBean和pku.as.service.naming.NamingServiceInterface接口，执行对pkuas命名服务的管理。

7.3.3.8. pku.as.naming.NamePaserImpl

命名解析器。

实现了javax.naming.NamePaser和java.io.Serializable接口。

以jndi语法为属性（从左到右分析、不忽略、消除空白、以'/'为分割）。

主要提供两个方法：

- getInstance() 返回一个命名解析器实例
- parse(String) 返回String的CompoundName名称

7.3.3.9. pku.as.naming.NamingServerExporter

这个类将pkuas命名服务器需要被外界调用的方法暴露给外界，提供一个pkuas监听的端口，从而实现RMI远程方法调用。

7.3.4. 其它相关类和接口说明

7.3.4.1. java.rmi.Remote

Remote 接口用于标识其方法可以从非本地虚拟机上调用的接口。任何远程对象都必须直接或间接实现此接口。只有在“远程接口”（扩展 java.rmi.Remote 的接口）中指定的这些方法才可远程使用。

扩展：

pku.as.naming.Naming

7.3.4.2. java.io.Serializable

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。序列化接口没有方法或字段，仅用于标识可序列化的语义。

实现：

- pku.as.naming.NamingServer
- pku.as.naming.SmartCtx
- pku.as.naming.NamePaserImpl

7.3.4.3. javax.naming.Context

此接口表示一个命名上下文，它由一组名称到对象的绑定组成，声明了一组上下文操作（查找、建立/重新/消除绑定、重命名、列表、创建/销毁子上下文）。

pkuas中的上下文是一颗树，树的节点是含有名称的对象。A/B/C这样的混合名称在命名树中的存储就是:A->B->C.命名树的逻辑存储方法是哈希表。

实现：

- pku.as.naming.SmartCtx
- pku.as.naming.ServerContextPlugin

7.3.4.4. javax.naming.spi.IntialContextFactory

此接口表示创建初始上下文的工厂。

JNDI 框架允许在运行时指定不同的初始上下文实现。初始上下文是使用初始上下文工厂创建的。初始上下文工厂必须实现InitialContextFactory 接口，该接口提供了一个方法，用于创建实现 Context 接口的初始上下文的实例。此外，工厂类必须是公共的，必须有一个不接受任何参数的公共构造方法。

实现：

- pku.as.naming.SmartCtxFactory
- pku.as.naming.ServerCtxFactory

7.3.4.5. javax.naming.NamePaser

此接口用于解析取自分层的名称空间的名称。NameParser 包含解析这些名称所需的语法信息知识（比如从左到右的方向、名称分隔符，等等）。 在使用 equals() 方法比较两个 NameParser 时，当且仅当它们服务于相同的名称空间时才返回 true。

实现：

- pku.as.naming.NamePaserImpl

7.3.4.6. javax.naming.NamingEnumeration

此接口用于枚举 javax.naming 和 javax.naming.directory 包中的方法所返回的非序列化的对象集合。

当某一方法（比如 list()、listBindings() 或 search()）返回 NamingEnumeration 时，在返回所有结果之前，将保留所遇到的所有异常。在枚举结束时抛出异常（由 hasMore() 完成）。

实现：

- pku.as.naming.NamingEnumerationImpl

7.3.4.7. pku.as.common.mbean.ServiceMBean

该接口提供对消息Bean的管理服务。

扩展：

- pku.as.naming.NamingServiceMBean

7.3.4.8. pku.as.service.naming.NamingServiceInterface

此接口提供创建企业级命名上下文的方法，也属于命名服务管理接口。

实现：

- pku.as.naming.NamingService

7.3.4.9. pku.as.common.ServiceBase

该抽象类扩展了javax.management.NotificationBroadcasterSupport类。

而javax.management.NotificationBroadcasterSupport提供NotificationEmitter 接口的实现。该类可以用作发送通知的 MBean 的超类。

默认情况下，通知调度模型是同步的。也就是说，当某一线程调用sendNotification 时，将在该线程中调用每个侦听器的NotificationListener.handleNotification 方法。可以通过重写子类中的handleNotification 或者通过将 Executor 传递给构造方法来重写此默认值。

如果过滤器或侦听器的方法调用抛出 Exception，则该异常不会阻止调用其他侦听器。不过，如果过滤器、Executor.execute 或 handleNotification 的方法调用（未指定任何 Excecutor 时）抛出 Error，则将该 Error 传播到sendNotification 的调用者。

通常不会同步调用使用 JMX Remote API 添加的远程侦听器。也就是说，当sendNotification 返回时，不保证任何远程侦听器都已经收到通知。

实现：

- pku.as.naming.NamingEnumerationImpl

第 8 章 安全服务

8.1. 安全模块概述

本模块为J2EE应用中EJB和WEB提供统一的认证和授权服务。整个应用（客户端构件与服务器端EJB构件和Web构件）使用统一的认证机制、安全域和角色映射。在独立接入方式中，安全检查在对服务器端的EJB构件进行调用时进行；在Web接入方式中，安全检查既可以在对Web页面访问时进行，也可以将检查延迟到对EJB构件访问时进行。

首先看该模块起作用的流程，下面这个例子：一个WEB客户依赖其WEB Server作为其认证代理。

step1.客户端访问WEB服务器的主应用URL，Web Server要求进行身份认证。

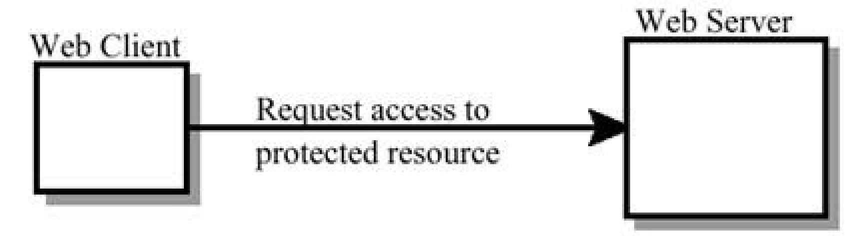


FIGURE 3-1 Initial Request

Step 2: Initial Authentication WEB server要求客户端填写并返还身份信息。

WEB server随后进行认证。WEB server的认证机制没有特殊要求，可以借助本地认证机制。最后必须生成一个身份凭证Credential。

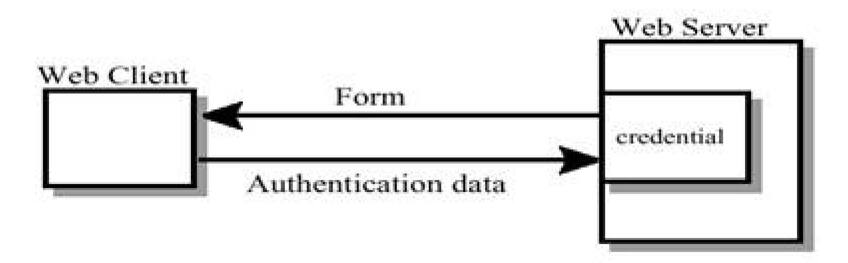


FIGURE 3-2 Initial Authentication

Step 3: URL Authorization Web server 访问安全策略，以决定是否允许该用户对其目标资源进行访问。Web container通过检查是否能将该用户和安全策略中定义的role对应实现. 若不能对应，则鉴权失败。

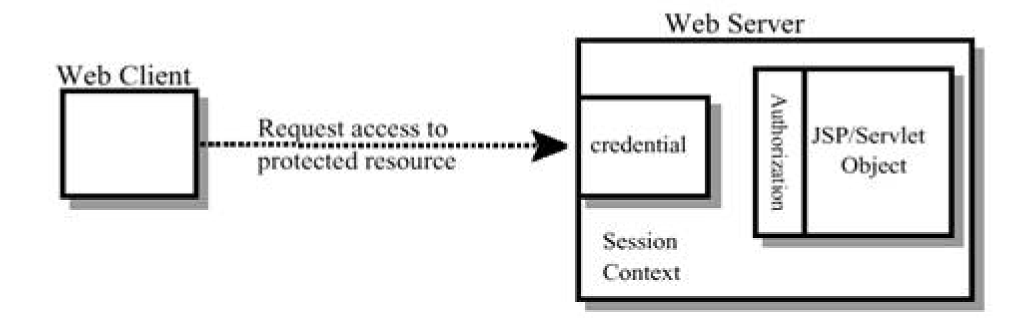


FIGURE 3-3 URL Authorization

Step 4: Fulfilling the Original Request 如果鉴权成功，WEB server将返回对初始URL的访问结果。之后，用户可能将进行业务操作。

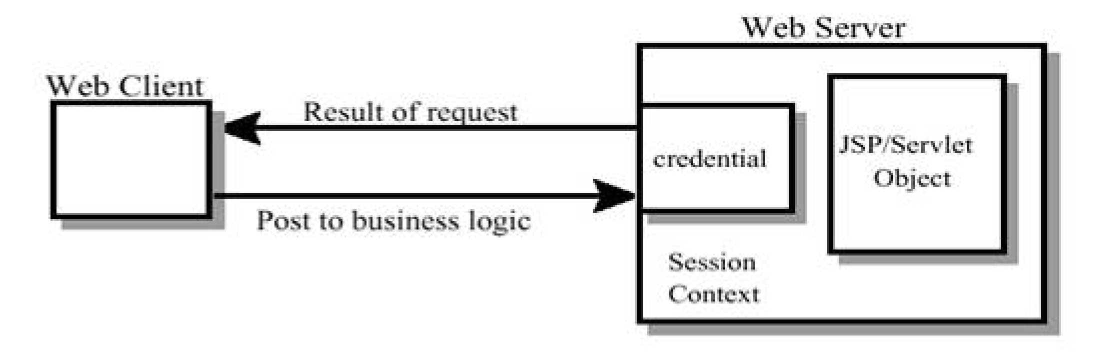


FIGURE 3-4 Fulfilling the Original Request

Step 5. Invoking Enterprise Bean Business Methods 当JSP或者Servlet需要对EJB进行远程调用时，用户的凭证将用于建立JSP/Servlet和EJB的安全关联。该关联通过Web Server和EJB Container双方的安全上下文实现。

EJB容器要为EJB的方法建立安全控制。它通过查询和该EJB关联的安全策略决定有权访问该方法的角色。这样对于每一个角色，EJB容器都会使用调用者的上下文以决定能否将调用者映射为角色。当鉴权成功后，容器将把控制转交给调用的EJB方法。

平台为需要显式进行安全控制的应用提供了两套接口： 对于EJB，通过EJBContext的isCallerInRole 和getCallerPrincipal方法。 对于servlet和JSP，通过HttpServletRequest的isUserInRole和getUserPrincipal 方法。 当EJB调用getCallerPrincipal时，EJB容器返回安全上下文中的Principal，该Principal不能为空。而HttpServletRequest的getUserPrincipal方法可以返回null.

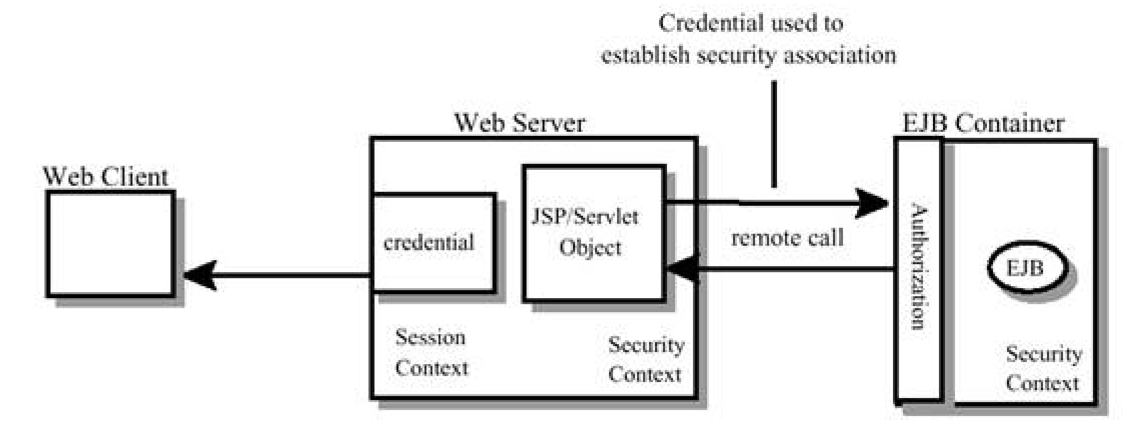


FIGURE 3-5 Invoking an Enterprise Bean Business Method

8.2. 安全体系结构

8.2.1. 基于容器的安全

在J2EE中，构件的安全通过它们的容器实现。容器通过两种途径提供安全：声明和程序接口。

声明型安全：

通过部署描述符描述。部署者将逻辑安全需求和物理的运行环境安全机制进行映射。

程序接口型安全：

如前所述，通过两个对象的4个API进行。

对于EJB，通过EJBContext的isCallerInRole 和getCallerPrincipal方法。

对于servlet和JSP，通过HttpServletRequest的isUserInRole和getUserPrincipal 方法。

8.2.2. 分布的安全

某些J2EE产品，可能不同的容器分布于网络上不同位置，这样构件间通信就引入安全问题。因此，要在构件间建立安全关联(共享的安全状态信息)，以保证通信的安全。建立过程如下：

- 为客户端认证目标，或反之。
- 协商安全质量。
- 建立构件间的安全上下文。

建立过程一般是由容器完成的。

8.2.3. 鉴权模型

J2EE鉴权模型基于安全角色的概念。 在部署描述符中的method-permission元素中，描述对方法的安全权限要求。

8.2.4. 角色映射

安全服务依赖于将请求的Principal和资源的安全角色的对应。

8.2.5. HTTP登录网关

J2EE 1.3版本没有定义不同安全策略域的互通。当访问其他J2EE产品时，可能不能互通。此时，构件可能选择先通过HTTP登录到外面的服务器。HTTP over SSL可以提供不同J2EE产品间的安全通信。我们称这种使用HTTP和SSL访问远端服务为HTTP登录网关(Login Gateway)。

8.2.6. 用户认证

根据终端用户的不同，有两种认证：基于Web客户的，和基于应用客户的。

8.2.6.1. web客户

Web 客户必须能够向Web Server认证用户。

WEB server可以使用如下认证机制：

HTTP Basic Authentication：要求客户输入名称和密码。没有加密，使用普通的Base64编码传输。可以通过某些方式加强，比如使用HTTP over SSL。

HTTPS Client Authentication：强认证机制，HTTP over SSL。用户必须拥有一个公钥证书。J2EE平台必须支持该机制。 Form Based Authentication：提供灵活的界面。

Web Single Signon

因为HTTP是无状态协议，但是很多WEB应用需要支持会话。因此，必须保证用户只需要认证一次，就可以访问所有的资源，只是在跨安全域时需要重新认证。容器使用凭证为会话建立安全上下文。

Login Session

在J2EE 平台中， 登录会话由Servlet容器支持。当用户成功认证后，容器为用户建立了登录会话，该会话包括用户的凭证信息。当客户是无状态时，服务器必须为他建立安全上下文，并通过某引用提供给客户使用。Cookie或者URL重写技术可以用于传递该引用。如果使用SSL,则不必依赖应引用。

8.2.6.2. 应用客户

指可以不通过Web Server就可以直接访问EJB的程序实体。常用于对客户的认证。

通常，只有当要访问被保护的资源时，才进行认证，称为延迟认证(Lazy Authentication)。

8.2.7. 用户认证需求

8.2.7.1. web客户

J2EE 产品提供商必须为Web 用户认证支持下述机制：

- Web Single Signon

  所有兼容J2EE的WEB server 必须支持Single Signon。必须支持一次登录会话跨越几个应用。

- Login Sessions

  所有J2EE产品必须支持Servlet规范中定义的登录会话。延迟认证也必须支持。

- Required Login Mechanisms

  三种机制必须被支持。

  HTTP Basic Authentication：包括HTTP over SSL。

  SSL Mutual Authentication：基于证书的认证。下述加密机制必须被支持：

  SSL_RSA_EXPORT_WITH_RC4_40_MD5 和 SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

- Unauthenticated Users

  WEB容器要为客户提供匿名访问身份。如果用户没有被认证，则HttpServletRequest的方法getUserPrincipal返回null。

  EJB规范要求调用EJBContext的方法getCallerPrincipal时，必须返回一个合法的Principal，永不能为null。 而WEB容 器内的构件必须能够访问EJB构件，即使其还没有任何用户通过认证。此时，J2EE产品必须提供一个Principal使用。

8.2.7.2. 应用客户

为了满足EJB容器和WEB容器的鉴权和认证要求，application client容器必须对应用的用户进行鉴权。进行鉴权的技术没有具体规定，可以集成J2EE产品的认证系统以提供single signon； 或者当应用启动时，容器就进行认证；或者延迟认证。

容器要提供一个适当的用户接口，以接受用户认证数据。

应用的回调句柄必须支持javax.security.auth.callback包中标识的所有Callback对象。

8.2.7.3. 对资源的认证需求

企业内的资源可能分布在不同的安全域中，尤其是可能不和构件处于一个安全域。J2EE产品必须能在资源所在的安全域中进行认证。下列机制被支持：

- Configured Identity.

  在部署时声明对资源进行访问的principal和相关的凭证数据。如果认证数据存储在容器中，则J2EE产品提供商负责保密性。必须支持部署时的指定，不能依靠程序代码。

- Programmatic Authentication.

  由应用构件在运行时提供principal和认证数据。应用构件可以通过多种方式获得该数据，包括通过参数，通过上下文等。

8.2.7.4. 鉴权需求

J2EE产品必须满足下列需求：

- Code Authorization

  J2EE产品可能限制使用某些J2SE类和方法，以保证系统安全。满足最低安全要求的应用必须能被部署。

- 对调用者鉴权

  必须支持调用者标识的传播。

对于一个J2EE产品的一个特定应用，其对所有EJB的调用，通过EJBContext的方法 getCallerPrincipal返回的Principal和调用链中的第一个EJB返回的Principal相同。如果调用链的第一个EJB是被servlet或者JSP调用的，则返回的Principal必须和HttpServletRequest的方法getUserPrincipal返回的Principal名称相同。该要求仅仅当EJB使用user-caller-identity时有效。

J2EE 产品也必须支持Run-AS特性。此时，初始调用者的身份不会被传递。

8.2.7.5. 部署需求

所有的J2EE产品必须实现EJB，JSP和Servlet规范中定义的安全控制语义，并且能将部署描述符中的逻辑定义和运行系统对应。J2EE产品必须提供部署工具，以将安全角色和运行时的负责安全的实体对应。

一个应用可能定义一个角色，代表所有认证的用户，或者没有认证的用户。

8.3. PKUAS的设计实现

安全服务作为一个系统服务，由pkuas加载启动。

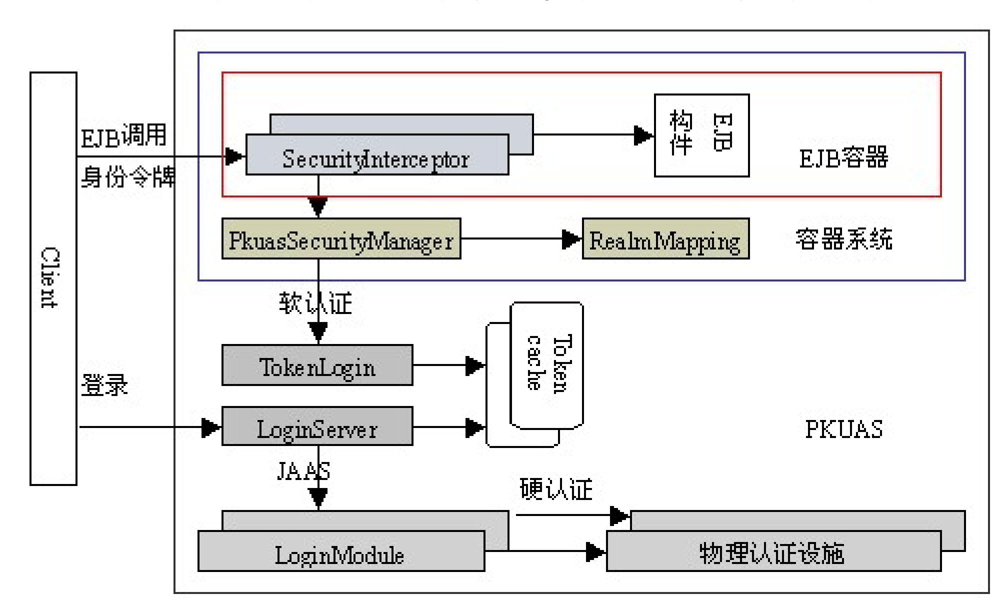
安全服务定义了SecurityServiceMBean接口，由SecurityService类实现该接口。

SecurityService类主要完成以下任务：

- 启动认证服务器LoginServer。
- 启动EJB安全管理器代理PkuasSecurityManagerProxy，该代理实现了远程访问PkuasSecurityManager的接口EJBSecurityManagerRemote，完成对EJB调用的安全检查。
- 启动安全设施管理器代理SecurityAdminProxy。该代理实现了远程管理底层安全认证机制的接口，可以远程增加、删除、修改用户。
- 集成符合JACC规范的授权策略模块，该模块完成应用服务器中的授权检查。应用服务器开发者在security.conf文件中声明授权策略提供商的相关参数，服务启动时，动态加载授权策略模块。

8.3.1. EJB容器的安全

通常，客户端对应用进行一次完整的业务操作流程，包含若干次WEB调用和EJB调用。这样一次完整的业务流程，就是一次“会话”的概念。由于EJB构件在处理收到的请求时，从请求本身得不到会话信息，这就迫使EJB构件对每一次EJB调用都执行安全检查。因此，EJB请求消息中，需要携带安全上下文，包括用户身份和凭证信息（通常是用户名和密码）。然而，在每次请求中都携带敏感的身份和凭证信息是不明智的。因此，在PKUAS的安全体系中，客户执行EJB业务操作前，要到认证服务器显式登录一次，获得一个令牌。该令牌是认证服务器分配给用户的临时身份凭证，代表一次会话。当用户注销登录，或者操作超时，该令牌就会失效。用户以后每次操作都携带该令牌，服务器检查操作请求中令牌的合法性。这样就降低了用户密码等安全信息的被窃取的概率。EJB容器的安全机制如图所示。



上图左边的Client可以是WEB客户端，也可以是Standalone客户端。PKUAS目前利用CORBA IIOP（Internet Interoperable ORB Protocol）消息传递EJB请求，IIOP消息头中特别为传输安全等上下文定义了消息域，安全信息保存在该消息域中。用户执行一次登录获得了令牌后，一个关键问题是如何透明地将安全上下文绑定到IIOP消息中。为了解决该问题，PKUAS为客户端提供了LoginClient模块，客户端只需要简单调用LoginClient提供的登录和注销方法即可。下图给出了

LoginClient执行登录请求的一段代码。在成功登录获得令牌后，将安全上下文设置到SecurityContext对象的线程相关变量中。当EJB客户桩Stub构造IIOP消息时，自动从SecurityContext中取出安全上下文。由于客户桩Stub是由工具自动生成的，因此安全上下文的绑定对用户而言是透明的。WEB客户端的调用略微复杂一些，在调用LoginClient的login后，还要建立令牌和HTTP会话的关联关系。在PKUAS中，令牌和HTTP会话之间关联关系的管理通过扩展Tomcat来实现。在WEB客户端发出EJB调用时， WEB容器会自动在IIOP消息中加入对应会话的安全上下文，对用户而言也是透明的。

```
public static Object login(String domain, Principal sp, Object passwd)
      throws LoginException, RemoteException
{     ……
      try { //variable ls is the remote instance of LoginServer
            token = ls.login(domain, sp, passwd) ;
      } catch (LoginException le) {
            throw le;
      } catch (RemoteException re) {
            throw re;
      }
      SecurityAssociation.setPrincipal(sp);
      SecurityAssociation.setCredential(token);
      return token ;
}
```

8.3.1.1. 安全截取器

在PKUAS中，EJB容器收到请求后，并不是直接将该请求转发给相应的EJB，而是要经过一系列的截取器（Interceptor），进行预处理。截取器机制体现了良好的“关注点分离”（Separation Of Concern）的思想，各种截取器可以完成不同的功能和服务，彼此却互不干扰；同时，截取器提供了对容器的行为进行扩展的能力——要增加新的行为，只需增加新的截取器。目前PKUAS中有三个主要的截取器：安全（ContainerSecurityInterceptor）、事务截取器（ContainerBMTInterceptor、ContainerCMTInterceptor）。

在PKUAS中，一个容器系统逻辑上对应一个客户应用，一个容器逻辑上对应一个应用构件。安全截取器是针对容器的，每个容器一个。安全截取器从EJB请求中取出安全上下文，即用户身份和令牌，交给SubjectSecurityManager进行认证和访问控制检查。

与安全截取器紧密相关的对象有安全管理器（PkuasSecurityManager）、域映射管理器（RealmMapping）、授权管理器（AuthorizationManager）。安全管理器和域映射管理器在一个容器系统范围内唯一；安全上下文中的用户身份是物理身份，在进行访问控制时，必须得到当前应用域为该物理身份定义的逻辑角色，根据逻辑角色判断访问权限。RealmMapping维护了应用安全域中物理用户和逻辑安全角色的映射关系，并提供访问接口。授权管理器负责管理J2EE应用中每个模块（ejb-jar、web-war）的访问控制。

当接受请求的EJB需要调用其它EJB时，调用者EJB传递的用户身份可以是该EJB持有的特定代理身份，即“run-as-specified-identify”方式；也可以将初始消息中的用户身份传递下去，即“use-caller-identity”方式，具体方式依赖于应用组装者在部署描述文件中的设置。安全截取器要负责处理运行时刻的相关工作。

8.3.1.2. 认证服务器

在PKUAS中，存在两种认证服务器：软认证服务器TokenLogin和物理认证服务器LoginServer。物理认证服务器LoginServer提供远程登录接口，负责处理客户的登录请求，根据JAAS的配置，调用相关的登录模块LoginModule，由相应的LoginModule负责实际物理认证。成功登录的用户会得到临时身份凭证—令牌Token。LoginServer维护了令牌缓存列表，记录令牌分配情况。软认证服务器TokenLogin只提供验证EJB调用中令牌合法性的本地接口，它只被PkuasSecurityManager调用。在实现上，TokenLogin和LoginServer是一个实现体。下图给出了TokenLogin和LoginServer的接口。

```
public interface TokenLogin
{
    public boolean verifyToken(String domain, Principal p, Object token)
        throws LoginException, RemoteException;
}
```

```
public interface LoginServer extends Remote
{
    public Object login(String domain, Principal p, Object credential)
        throws LoginException, RemoteException;
    public void logout(String domain, Principal p, Object token)
        throws RemoteException;
}
```

通过上述安全机制，PKUAS实现了J2EE规范的基本安全需求。PKUAS数据传输的安全是通过安全套接层协议SSL实现的，利用了JSSE提供的API和参考实现，在此不再详述。

在JAAS配置文件中说明了应用的安全域jwdomain，配置用于进行认证的认证对象：LoginModule。EJB容器有一个安全截取器，首先截取了Comm模块发送给EJB容器的调用请求，授权给和该容器关联的安全管理器进行认证。

对于缺省的安全管理器JaasSecurityManager，它的认证基于从消息中提取的Principal对象和Credential对象。通过执行配置的LoginModule，得到认证结果。

8.3.1.3. 安全管理器SubjectSecurityManager

SubjectSecurityManager继承自EJBSecurityManager，提供了本地调用接口，SecurityManagerRemote提供了远程访问接口。

本地访问接口SubjectSecurityManager，提供给ContainerSecurityInterceptor使用。远程接口SecurityManagerRemote，提供给WEB服务器使用。因为WEB服务器使用和EJB容器相同的认证机制，而它有可能分布在别处，不和EJB容器在一个JVM中，因此，需要一个远程访问接口。

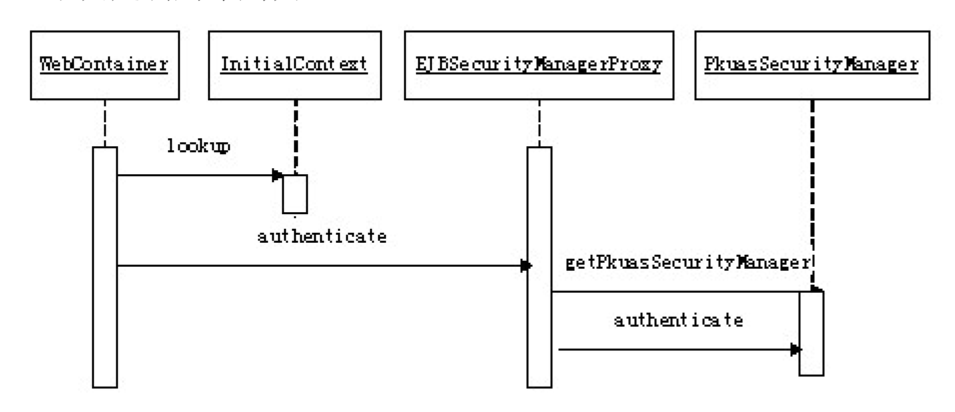
实际中，PkuasSecurityManager类实现了SubjectSecurityManager接口，负责安全检查。PkuasSecurityManager调用软认证服务器TokenLogin进行软认证（所谓软认证，是指没有在实际的安全基础设施中进行物理认证，只是对令牌进行认证调用）；调用EJBAuthorizationManager进行访问控制。PkuasSecurityManager同时

还为EJB编程型安全模型提供支持，EJBContext的接口最后都是通过PkuasSecurityMananger获得相关信息。

远程访问通过代理SecurityManagerProxy实现。SecurityManagerProxy实现了SecurityManagerRemote接口，把对SecurityManagerRemote的调用，转化为对相应PkuasSecurityManager实例的调用。SecurityManagerProxy由安全服务启动，它控制所有应用的远程安全检查，根据调用参数，调用相应的PkuasSecurityManager实例。

SecurityManagerRemote的类定义如下，其中各个方法调用中的domain参数，标识了应用安全域，应用安全域在应用描述文件application.xml中由security-domain项配置。

下面是调用序列图



8.3.1.4. 授权管理器AuthorizationManager

PKUAS中的授权遵循JACC规范，为了更好地进行访问权限的控制，每个jar包和war包分别拥有一个授权管理器EJBAuthorizationManager，WebAuthorizationManager，这两个类集成自抽象类AuthorizationManager。它们的主要职责为：

- 将部署描述符中的有关访问权限的安全声明转换成JACC规定Permission，存放在JACC的PolicyConfiguration中，每个jar包或war包对应一个PolicyConfiguration。
- 管理PolicyConfiguration状态的改变（commit、remove）。
- 进行访问控制检查，事实上授权管理器将访问检查工作交给JACC授权策略模块中的Policy类来完成。　　依据JACC的要求，PKUAS自己实现了一个JACC授权策略模块，完成授权的基本功能。改模块包括类PkuasPolicy、PkuasPolicyConfiguration、PkuasPolicyConfigurationFactory。

8.3.2. web容器的安全

一般来说，Web容器的安全重点在控制资源的访问上，但也有一些程序将资源访问的认证与授权和业务逻辑相统一。从PKU-AS的整体考虑，希望能够让用户在整个访问过程中只进行一次登录，这就需要将Web容器的安全和EJB容器的安全统一起来。

Web容器中的安全基于Tomcat的实现，只考虑Servlet引擎部分的安全。这主要是因为从现实情况看，将Tomcat直接作为Web Server并不合适，这就需要PKU-AS能够保持与Web Server之间的独立性，从而保证使用PKU-AS的用户可以根据自己需要选择合适的Web Server。

PKU-AS中Web容器的安全通过修改集成的Tomcat实现。保证：

- 修改后的Tomcat能够符合Servlet规范的规定，实现2.3中提到的标准的身份认证与授权，完全符合规范的应用程序可以不加修改的部署、运行于PKU-AS而不用修改其安全设定。
- 修改后的Tomcat作为为PKU-AS统一的安全认证与授权的一部分，保持与后端EJB Server的一致性。
- 具体实现方式为自定义安全截取器（Security Interceptor）来对用户进行认证和授权。
- Tomcat中不做独立的LoginModule，只是作为一个认证授权的点存在，实际的工作通过调用SecurityManager接口而实现。Tomcat只完成用户信息向SecurityManager的传递、返回认证授权信息（如Pricipal、Cridential）与session的绑定，以及之后每次请求与对应安全上下文的映射。

当客户输入一个网络资源地址、或通过超级链接连向一个网络资源，客户将向WebSever（web容器）发出一个http的客户请求（HttpRequest）。这里，我们只讨论客户将访问一个受保护的资源的情况。

在Tomcat中，有两个RequestInterceptor — AccessInterceptor、SimpleRealmInterceptor — 处理身份认证与授权。为保证PKU-AS对Servlet标准规定的身份认证与授权的支持，暂不改动这两个Interceptor及其在Tomcat中的位置。 从总的过程来看，web容器将向客户发出要求身份认证的请求，要求用户输入用户名和密码，再由web容器根据用户信息进行身份认证和授权。当用户通过认证并得到授权后，web容器正常服务；否则返回出错页面。

第 9 章 事务服务

9.1. 第1小节标题

9.1.1. 第1.1小节标题

第1.1小节内容

9.1.2. 第1.2小节标题

第1.2小节内容

9.2. 第2小节标题

9.2.1. 第2.1小节标题

第2.1小节内容

9.2.2. 第2.2小节标题

第2.2小节内容

第 10 章 JMS 服务

10.1. 第1小节标题

10.1.1. 第1.1小节标题

第1.1小节内容

10.1.2. 第1.2小节标题

第1.2小节内容

10.2. 第2小节标题

10.2.1. 第2.1小节标题

第2.1小节内容

10.2.2. 第2.2小节标题

第2.2小节内容

第 11 章 数据服务

11.1. 第1小节标题

11.1.1. 第1.1小节标题

第1.1小节内容

11.1.2. 第1.2小节标题

第1.2小节内容

11.2. 第2小节标题

11.2.1. 第2.1小节标题

第2.1小节内容

11.2.2. 第2.2小节标题

第2.2小节内容

第 12 章 持久服务

12.1. 第1小节标题

12.1.1. 第1.1小节标题

第1.1小节内容

12.1.2. 第1.2小节标题

第1.2小节内容

12.2. 第2小节标题

12.2.1. 第2.1小节标题

第2.1小节内容

12.2.2. 第2.2小节标题

第2.2小节内容

第 13 章 日志服务

PKUAS集成了log4j作为其日志系统，同时也引入了commons-logging。采用的是二者结合的流行模式。日志部分的技术白皮书将首先对log4j，commons-logging和两者的结合进行介绍，然后具体介绍在PKUAS-2010中的日志使用方法和示例。

13.1. 日志服务软件包介绍

13.1.1. log4j简介

Log4j是Apache的一个开放源代码项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI组件、甚至是套接口服务器、NT的事件记录器、UNIX Syslog守护进程等；我们也可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程。这些可以通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。

Log4j主要由三大类组件构成：

1）Logger

负责输出日志信息，并能够对日志信息进行分类筛选，即决定哪些日志信息应该被输出，哪些该被忽略。Loggers组件输出日志信息时分为5个级别：DEBUG、INFO、WARN、ERROR、FATAL。这五个级别的顺序是：

DEBUG<INFO<WARN<ERROR<FATAL

如果设置某个Logger组件的级别是P，则只有级别比P高的日志信息才能输出。Logger是有继承关系的，最上层是rootLogger，定义的其他Logger都会继承rootLogger。

2）Appender

定义了日志输出目的地，指定日志信息应该被输出到什么地方。输出的目的地可以是控制台、文件或网络设备。

3）Layout

通过在Appender的后面附加Layout来实现格式化输出。

一个Logger可以有多个Appender，每个Appender对应一个Layout。

13.1.2. commons-logging

Commons-logging在org.apache.commons.logging包中，它是对多种日志APIs的简单封装包。该包为服务器端程序的日志处理提供API以使用多种不同的日志系统。包括如下已经实现的：

- Log4J: Apache Jakarta 项目。每个Log的实例都对应于一个Log4j Category类。
- JDK: Logging API JDK1.4及后续版本中。每个Log的实例都是一个java.util.logging.Logger实例。
- LogKit: Apache Jakarta 项目。每个Log的实例都对应于一个LogKit Logger类。
- NoOpLog: 简单地接受将所有的Log实例的日志输出，。
- SimpleLog: 将所有的Log实例的日志输出到 System.out中。

13.1.3. commons-logging和Log4j的结合

由于Commons-logging的目的是为“所有的Java日志实现”提供一个统一的接口，它自身的日志功能很弱（只有一个简单的SimpleLog），所以一般不会单独使用它。Log4j的功能非常全面强大，是目前的首选。几乎所有的Java开源项目都会用到Log4j，而所有用到Log4j的项目一般也同时会用到commons-loggin。原因之一是不希望项目与log4j绑定得太紧密，并且，同时使用commons-logging和Log4j可以简化使用和配置。

Commons-logging提供一个统一的日志接口，简单了操作，同时避免项目与某个日志实现系统（如log4j）紧密耦合。它可以自动选择适当的日志实现系统，甚至不需要配置。

自动选择日志实现系统的机制如下：

1） 首先在classpath下寻找自己的配置文件commons-logging.properties，如果找到，则使用其中定义的Log实现类；

2） 如果找不到commons-logging.properties文件，则再找是否已定义系统环境变量org.apache.commons.logging.Log，找到则使用其定义的Log实现类；

3） 否则，查看classpath中是否有Log4j的包，如果发现，则自动使用Log4j作为日志实现类；

4） 否则，使用JDK自身的日志实现类；

5） 否则，使用commons-logging自己提供的一个简单的日志实现类SimpleLog。

可见， commons-logging总是能找到一个日志实现类，并且尽可能找到一个“最合适”的日志实现类。这样的机制给开发者带来方便，因为：

1）可以不需要配置文件；

2）自动判断有没有Log4j包，有则自动使用之；

3）最悲观的情况下也总能保证提供一个日志实现（SimpleLog）。

可以看到，commons-logging对编程者和Log4j都非常友好。

为了简化配置，一般不使用commons-logging的配置文件，也不设置与commons-logging 相关的系统环境变量，而只需将Log4j的Jar包放置到classpash

中就可以了。这样就很简单地完成了commons-logging与 Log4j的融合。如果不想用Log4j了，只需将classpath中的Log4j的Jar包删除即可。

13.2. commons-logging和log4j结合下日志的使用

一般而言，在需要输出日志信息的类中做如下工作：

1、导入所需的commons-logging类。

2、在使用日志的类中定义一个org.apache.commons.logging.Log类的私有静态类成员。

该成员将获取日志记录器，这个记录器将负责控制日志信息。这将通过指定的名字获得记录器，如果必要的话，则为这个名字创建一个新的记录器。

3、使用org.apache.commons.logging.Log类的成员方法输出日志信息。

当上两个必要步骤执行完毕，就可以轻松地使用不同优先级别的日志记录语句插入到想记录日志的任何地方。

以上所介绍的方法是目前被普通应用的，可以说是被标准化了的方法，很多知名的Java开源项目源代码中就是这样使用的。

要注意的几点：

1）需要将commons-logging和log4j的jar包放到classpath中，commons-logging才会自动发现并应用Log4j；

2）配置文件log4j.properties对Log4j来说是必须的。如果classpath中没有该配置文件，或者配置不对，将会引发运行时异常；

3）如果不用commons-logging，仅仅单独使用Log4j，操作上反而要稍微麻烦一些，因为Log4j需要多一点的初始化代码（相比commons-logging而言），所以引用commons-logging并没有使问题复杂化，反而简单了一些（在这里1+1 < 2了）。

总之，commons-logging提供了简捷、统一的接口，不需要额外配置，简单；Log4j功能非常全面、强大；因此，采用Log4j配合commons-logging作为日志系统，成为目前Java领域非常流行的模式，使用非常普遍。两者的结合带来的结果就是：简单 + 强大。

13.3. log4j的配置方法

Log4j支持两种配置文件格式,一种是XML格式的文件,一种是配置Java特性文件(键=值)，即log4j的properties文件。

13.3.1. java特性文件配置方法

下面通过分析一个配置文件log4j.properties介绍使用Java特性文件做为配置文件的方法。

```
  #1
# 配置根logger
#### Use two appenders, one to log to console, another to log to a file
log4j.rootLogger = debug, stdout
#2
#Print only messages of priority WARN or higher for your category
log4j.logger.cn.edu.pku.TestLog4j= , R
log4j.logger.cn.edu.pku.TestLog4j.TestLog4j2=WARN
#3
#### First appender writes to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n
#4
#### Second appender writes to a file
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=example.log
# Control the maximum log file size
log4j.appender.R.MaxFileSize=100KB
# Archive log files (one backup file here)
log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%d{yyyy-MM-dd hh:mm:ss}:%p %t %c - %m%n
```

说明:

（1）配置根Logger，其语法为：

```
log4j.rootLogger = [ level ] , appenderName, appenderName, …
```

其中，level 是日志记录的优先级，分为OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL或者自定义的级别。Log4j建议只使用四个级别，优先级从高到低分别是ERROR、WARN、INFO、DEBUG。通过在这里定义的级别，您可以控制到应用程序中相应级别的日志信息的开关。比如在这里定义了INFO级别，只有等于及高于这个级别的才进行处理，则应用程序中所有DEBUG级别的日志信息将不被打印出来。ALL:打印所有的日志，OFF：关闭所有的日志输出。appenderName就是指定日志信息输出到哪个地方，可同时指定多个输出目的地。

（2）配置日志信息输出目的地 Appender

Appender 负责控制日志记录操作的输出，其语法为：

```
log4j.appender.appenderName = fully.qualified.name.of.appender.class
log4j.appender.appenderName.option1 = value1
…
log4j.appender.appenderName.optionN = valueN
```

这里的appenderName为在①里定义的，可任意起名。

其中，Log4j提供的appender有以下几种：

org.apache.log4j.ConsoleAppender（控制台）

org.apache.log4j.FileAppender（文件）

org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件）

org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件），可通过log4j.appender.R.MaxFileSize=100KB设置文件大小，还可通过log4j.appender.R.MaxBackupIndex=1设置为保存一个备份文件

org.apache.log4j.WriterAppender（将日志信息以流格式发送到任意指定的地方）

（3）配置日志信息的格式（布局）Layout

Layout负责格式化Appender的输出，其语法为：

```
log4j.appender.appenderName.layout = fully.qualified.name.of.layout.class
log4j.appender.appenderName.layout.option1 = value1
…
log4j.appender.appenderName.layout.optionN = valueN
```

其中，Log4j提供的layout有以下几种：

org.apache.log4j.HTMLLayout（以HTML表格形式布局）

org.apache.log4j.PatternLayout（可以灵活地指定布局模式）

org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串）

org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等等信息）

（4）格式化日志信息

Log4J采用类似C语言中的printf函数的打印格式格式化日志信息，打印参数如下：

%m 输出代码中指定的消息

%p 输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL

%r 输出自应用启动到输出该log信息耗费的毫秒数

%c 输出所属的类目，通常就是所在类的全名

%C输出所在类的全名

%t 输出产生该日志事件的线程名

%n 输出一个回车换行符，Windows平台为“rn”，Unix平台为“n”

%d 输出日志时间点的日期或时间，默认格式为ISO8601，也可以在其后指定格式，比如：%d{yyyy MMM dd HH:mm:ss,SSS}，输出类似：2002年10月18日 22：10：28，921

%l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数

%L输出日志事件的发生位置在代码中的行数

%F输出日志事件的发生的文件名

（5）应用记录器的层次

任何一个记录器的使用都有两个步骤：

Step1. 在配置文件中定义相应的记录器；

Step2. 在代码中调用Logger类的取得记录器方法取得相应的记录器对象。

在配置文件中定义记录器的格式有两种，定义根记录器的格式为：

```
log4j.rootLogger = [ level ], appendName1, appendName2, …appendNameN
```

定义一个非根记录器的格式为：

```
log4j.logger.loggerName1 = [ level ], appendName1,…appendNameN
……
log4j.logger.loggerNameM = [ level ], appendName1, …appendNameN
```

Logger是有继承关系的，最上层是rootLogger，定义的其他Logger都会继承rootLogger，其他logger之间也可以有继承关系。上述文件，1区中定义了一个根记录器，具有DEBUG级别并有一个名称为stdout的输出端appender；2区应用到了记录器层次。在这里，定义了两个名称分别为TestLog4j和TestLog4j.TestLog4j2设计器。

在定义TestLog4j记录器时没有指定级别，所以继承自它的父记录器，即根记录器，同时又定义了一个名称为R的输出端，所以它的输出端有两个，一个从根记录器继承而来的名为stdout的输出端，另一个为在此定义的名为R的输出端。（在此需要注意的是，在定义记录器时必须先定义记录器的级别，然后才是输出端，如果只想定义输出端，则逗号分隔符不能省略。）

在定义TestLog4j.TestLog4j2记录器时指定了它的级别，一个记录器的级别只能有一个，所以覆写掉它的父记录器的级别，同时也没有定义TestLog4j.TestLog4j2记录器的输出端，所以也从父记录器继承。注意，由于起名字的方式，它的父记录器为estLog4j记录器，所以它也有两个输出端。

要取得根记录器对象可通过Logger.getRootLogger()函数，要取得非根记录器可通过Logger.getLogger()函数。

13.3.2. xml格式的log4j配置文件概述

xml格式的log4j配置文件需要使用org.apache.log4j.html.DOMConfigurator.configure()方法来读入。对xml文件的语法定义可以在log4j的发布包中找到org/apache/log4j/xml/log4j.dtd.

下面通过分析两个配置文件sample1.xml与sample2.xml来介绍如何配置xml格式的log4j配置文件。

文件sample1.xml

```
<?xml version="1.0" encoding="GB2312" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

<appender name="FILE"  class="org.apache.log4j.RollingFileAppender"> <!-- 设置通道
  <param name="File" value=" all.output.log" /><!-- 设置File参数：日志输出文件名 -
  <param name="Append" value="true" /><!-- 设置是否在重新启动服务时，在原有日志的基
  <param name="MaxBackupIndex" value="10" />
  <layout class="org.apache.log4j.PatternLayout">
   <param name="ConversionPattern" value="%p (%c:%L)- %m%n" /><!-- 设置输出文件项目
  </layout>
 </appender>

 <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
              <!--设置控制台输出方式-->
  <layout class="org.apache.log4j.PatternLayout">
   <param name="ConversionPattern"
    value="%-4r [%t] %-5p %c %x - %m%n" />
```

```
  </layout>
  <filter class="org.apache.log4j.varia.LevelRangeFilter"> <!--滤镜设置输出的级别-
   <param name="levelMin" value="info" />
   <param name="levelMax" value="info" />
   <param name="AcceptOnMatch" value="true" />
  </filter>
 </appender>

 <root><!-- 设置接收所有输出的通道 -->
  <priority value="info" />
  <appender-ref ref="FILE" />
  <appender-ref ref="STDOUT" /><!-- 与前面的通道id相对应 -->
 </root>

</log4j:configuration>
```

说明:

①xml配置文件的头部包括两个部分:xml声明和dtd声明.

②log4j:configuration (root element) xmlns:log4j [#FIXED attribute]: 定义log4j的名

appender [* child] : 一个appender子元素定义一个日志输出目的地

logger [* child] : 一个logger子元素定义一个日志写出器

root [child] : root子元素定义了root logger

另一个xml范例文件对一些元素和属性进行了更加详尽的讲解，可以参考。

文件sample2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/" debug="false">
    <!--输出方式：输出到控制台-->
    <appender name="myConsole" class="org.apache.log4j.ConsoleAppender">
    <!--
     设置通道名称和输出方式，其中输出方式appender有5种,分别为
      org.apache.log4j.RollingFileAppender
     org.apache.log4j.ConsoleAppender (控制台)
     org.apache.log4j.FileAppender (文件)
     org.apache.log4j.DailyRollingFileAppender (每天产生一个日志文件)
     org.apache.log4j.WriterAppender (将日志信息以流格式发送到指定地方)
     -->
    <param name="Target" value="System.out"/>
    <param name="Threshold" value="INFO"/>
    <!--Threshold是个全局的过滤器，他将把低于所设置的level的信息过滤不显示出来 -->
    <!-- 设置日志输出的格式-->
<layout class="org.apache.log4j.PatternLayout">
<!--
```

```
        参数都以%开始后面不同的参数代表不同的格式化信息：
        %c  输出所属类的全名，可在修改为 %d{Num} ,Num类名输出的范围
        如："org.apache.elathen.ClassName",%C{2}将输出elathen.ClassName
%C输出所在类的全名
       %d  输出日志时间其格式为 %d{yyyy-MM-dd HH:mm:ss,SSS}，可指定格式 如 %d{HH:mm:
       %l  输出日志事件发生位置，包括类目名、发生线程，在代码中的行数
        %n  换行符
        %m  输出代码指定信息，如info(“message”),输出message
        %p  输出优先级，即 FATAL ,ERROR 等
        %r  输出从启动到显示该log信息所耗费的毫秒数
        %t  输出产生该日志事件的线程名
%L输出日志事件的发生位置在代码中的行数。
%F输出日志事件的发生的文件名。

-->

       <!-- The default pattern: Date Priority [Category] Message\n -->
        <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
      </layout>
    </appender>
    <!--输出方式是：每天一个日志文件-->
    <!--设置通道名称是：file，输出方式DailyRollingFileAppender-->
    <appender name="myFile" class="org.apache.log4j.DailyRollingFileAppender">
     <!--日志文件路径和文件名称 -->
      <!-- 如果在加载时设置了变量System.setProperty("WebApp", appRoot)，可在此取出来
      <param name="File" value="../logs/mylog.log"/>
      <!-- 设置是否在重新启动服务时，在原有日志的基础添加新日志 -->
      <param name="Append" value="true"/>
      <!-- Rollover at midnight each day -->
      <!-- e.g. mylog.log.2009-11-25.log -->
      <param name="DatePattern" value="'.'yyyy-MM-dd'.log'"/>
     <!-- Rollover at the top of each hour
      <param name="DatePattern" value="'.'yyyy-MM-dd-HH'.log'"/>
      -->
     <layout class="org.apache.log4j.PatternLayout">
       <!-- The default pattern: Date Priority [Category] Message\n -->
        <param name="ConversionPattern" value="%d %-5p [%c] %m%n"/>
<!-- The full pattern: Date MS Priority [Category] (Thread:NDC) Message\n -->
        <!--
        <param name="ConversionPattern" value="%d %-5r %-5p [%c] (%t:%x) %m%n"/>
         -->
      </layout>
    </appender>

    <appender name="ERROR_LOG" class="org.apache.log4j.DailyRollingFileAppender">
        <errorHandler class="org.apache.log4j.helpers.OnlyOnceErrorHandler"/>
        <param name="File" value="error.log"/>
        <param name="Append" value="true"/>
```

```
        <!-- 指定日志输出级别 -->
        <param name="Threshold" value="INFO"/>
        <param name="DatePattern" value="'.'yyyy-MM-dd'.log'"/>
        <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d %-5p [%c] %m%n"/>
        </layout>
     </appender>


  <!--
          level:是日记记录的优先级，优先级由高到低分为
            OFF ,FATAL ,ERROR ,WARN ,INFO ,DEBUG ,ALL。
            Log4j建议只使用FATAL ,ERROR ,WARN ,INFO ,DEBUG这四个级别。
       -->
      <!-- 指定logger的设置，additivity指示是否叠加输出log，如果是false，
        在DsErrorLog logger中日志不会被其它logger满足条件的logger(比如root)
       输出
      -->
      <!-- 将名称为DSErrorLog的logger，输出到“EEROR_LOG”的appender
           所谓logger的名字也就是，在定义Logger时，构造函数的参数
            Logger log = Logger.getLogger("DSErrorLog");
      -->
      <logger name="DSErrorLog" additivity="false">
          <level class="org.apache.log4j.Level" value="DEBUG"/>
          <appender-ref ref="ERROR_LOG"/>
      </logger>

      <!--
        输出指定类包中的日志，比如想输出
         Hibernate运行中生成的SQL语句，可作如下设置
      -->
      <category name="org.hibernate.SQL">
        <priority value="DEBUG"/>
        <!--
            如果指定一个appender,这些log将被输出到指定的appender
            如：<appender-ref ref="myFile"/>
            否则将作用于所有的appender
         -->
      </category>

       <!-- 根默认会自动构建一个 root,输出INFO级别的日志到控制台，供logger继承 -->
       <root>
           <priority value ="INFO"/>
           <appender-ref ref="myConsole"/>
           <appender-ref ref="myFile"/>
  </root>
  </log4j:configuration>
```

13.3.3. 补充

在配置文件中，logger定义的日志级别可以和appender的不一样。在定义logger时，尽量把日志级别往低里写；在appender里定义日志级别时可以按需要来写。

13.4. PKUAS-2010中的日志服务

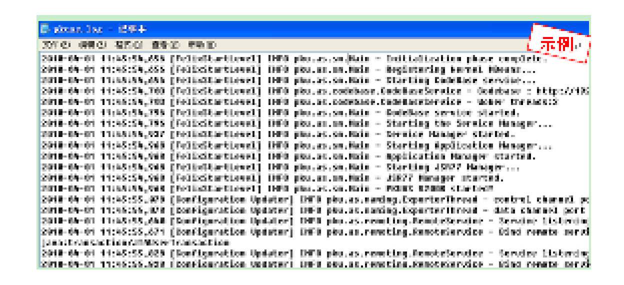
13.4.1. 概述

日志服务属于各功能模块的共性元素，更详细地说是属于共性元素中所有模块的AOP元素(所谓AOP服务，指的是日志、CodeBase这样的内核服务)，根据PKUAS2006的体系结构，它由微内核提供，在重构为PKUAS2008时，关于模块共性元素的大部分代码基本没有进行修改。

具体而言，PKUAS集成了log4j作为其日志系统，同时也引入了commons-logging。因此采用的是二者结合的流行模式。

执行run.bat启动编译生成的可运行版本，其启动过程将被记录在%PKUAS_HOME%\logs\下的pkuas.log和pkuas-web.log这两个相应的日志文件中（PKUAS_HOME为可运行版本解压的根目录）。如下显示的是pkuas.log文件示例：

图 13.1. %PKUAS_HOME%\logs\pkuas.log文件概要示例

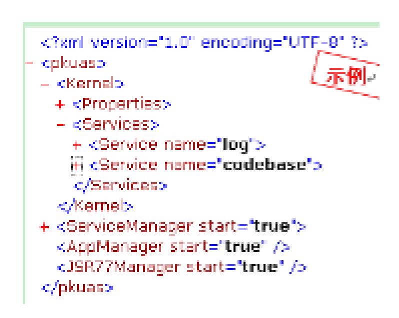


13.4.2. 日志服务的配置示例

由上述材料可知，log4j有最基本的两种配置方法，熟知的是配置properties文件,另一种是采用xml配置的方法。PKUAS2006版本中，由文件pkuas.log4j负责对PKUAS的日志进行配置，但在PKUAS-OSGi-2010中，日志服务和其他功能模块的配置都被放到了pkuas.xml文件中。

查阅%PKUAS_HOME%\conf\下的pkuas.xml文件，概略如下：

图 13.2. pkuas.xml文件概要示例



```
<?xml version="1.0" encoding="UTF-8" ?>
- <pkuas>
 - <Kernel>
  + <Properties>
  - <Services>
   + <Service name="log">
   + <Service name="codebase">
   </Services>
  </Kernel>
 + <ServiceManager start="true">
  <AppManager start="true" />
  <JSR77Manager start="true" />
 </pkuas>
```

可以发现，log和codebase是作为内核的两个服务进行配置的，而ServiceManager下则是对pkuas各个功能模块的配置。

下面详细看看其中对log服务的配置情况。

图 13.3. Log服务的具体配置示例

```
- <Services>
 - <Service name="log">
  - <log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
   - <appender name="dummy" class="org.apache.log4j.ConsoleAppender">
    - <layout class="org.apache.log4j.PatternLayout">
       <param name="ConversionPattern" value="%d{HH:mm:ss}[%t]%c{1}(%p) %m%n" />
      </layout>
     </appender>
   - <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
    - <layout class="org.apache.log4j.PatternLayout">
       <param name="ConversionPattern" value="%d{HH:mm:ss}[%t]%c{1}(%p) %m%n" />
      </layout>
     </appender>
   - <appender name="PkuasLogFile" class="org.apache.log4j.RollingFileAppender">
      <param name="File" value="../logs/pkuas.log" />
      <param name="Append" value="false" />
      <param name="MaxFileSize" value="300KB" />
      <param name="MaxBackupIndex" value="10" />
    - <layout class="org.apache.log4j.TTCCLayout">
       <param name="DateFormat" value="ISO8601" />
      </layout>
     </appender>
   - <appender name="WebLogFile" class="org.apache.log4j.RollingFileAppender">
      <param name="File" value="../logs/pkuas-web.log" />
      <param name="Append" value="false" />
      <param name="MaxFileSize" value="300KB" />
      <param name="MaxBackupIndex" value="10" />
    - <layout class="org.apache.log4j.TTCCLayout">
       <param name="DateFormat" value="ISO8601" />
      </layout>
     </appender>
   - <root>
      <priority value="info" />
      <appender-ref ref="dummy" />
     </root>
   - <category name="pku" additivity="false">
      <priority value="INFO" />
      <appender-ref ref="STDOUT" />
      <appender-ref ref="PkuasLogFile" />
     </category>
   - <category name="org.apache" additivity="false">
      <priority value="info" />
      <appender-ref ref="STDOUT" />
      <appender-ref ref="WebLogFile" />
     </category>
    </log4j:configuration>
   </Service>
```

说明：

此示例文件为用于log4j的xml形式的配置文件，它一共指定了4个输出器（appender ）：

1. dummy

输出到控制台；

2. STDOUT

输出到控制台（同dummy没有区别）；

3.PkuasLogFile

RollingFileAppender, 即滚动日志文件，具体记录在../logs/pkuas.log文件中；文件大小到达指定尺寸300KB的时候产生一个新的文件；最多有10个备份；TTCCLayout，包含日志产生的时间、线程、类别等等信息；在重新启动服务时，不会在原有日志的基础添加新日志，而是覆盖掉原有内容；

4.WebLogFile

RollingFileAppender，同上，滚动日志文件，具体记录在../logs/pkuas-web.log文件中；文件大小到达指定尺寸300KB的时候产生一个新的文件；最多有10个备份；TTCCLayout，包含日志产生的时间、线程、类别等等信息；在重新启动服务时，不会在原有日志的基础添加新日志，而是覆盖掉原有内容；

接着指定了以下几个Logger，包括rootLogger和两个包的category：

1.rootLogger

info级别，设置接收上面命名为dummy的输出通道，即输出到控制台；

2.pku

输出指定包pku中的日志，指定这些日志被输出到STDOUT和PkuasLogFile这两个输出通道;additivity指示是否叠加输出log，此处设置成false，则不会被其它满足条件的logger输出(比如root)；

3.org.apache

输出指定包org.apache中的日志,指定这些日志被输出到STDOUT和WebLogFile这两个输出通道；同样的，additivity设置成false，不会被其它满足条件的logger输出。

通过分析控制台内容及logs目录下的日志文件即可。由tomcat相关的包产生的日志由log4j管理，输出到pkuas-web.log，pkuas自己的代码产生的日志输出到pkuas.log，同时两者都会全部输出到控制台。

13.4.3. 日志使用方法

下面以pkuas中日志的使用为例，总结一下日志的使用方法。一般而言，在需要输出日志信息的类中做如下工作：

1、导入所有需的log4j类

Pkuas中的例子：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

2、得到记录器(Logger)

获取日志记录器，这个记录器将负责控制日志信息。这将通过指定的名字获得记录器，如果必要的话，则为这个名字创建一个新的记录器。Name一般取本类的名字，比如，在要使用日志服务的类中，声明:

Pkuas中的例子：

```
public final class Main {
private static Log logger = LogFactory.getLog(Main.class);
……
}
```

注意这里定义的是static成员，以避免产生多个实例，LogFactory.getLog()方法的参数使用的是当前类的class，这是目前被普通认为的最好的方式。（注意不写作LogFactory.getLog(this.getClass()，因为static类成员访问不到this指针)

3、输出日志信息

当上两个必要步骤执行完毕，就可以轻松地使用不同优先级别的日志记录语句插入到想记录日志的任何地方，其语法如下：

```
logger.debug ( Object message ) ;
logger.info ( Object message ) ;
logger.warn ( Object message ) ;
logger.error ( Object message ) ;
logger.fatal(Object message);
```

这里的logger是第2步中定义的类成员变量，其类型是org.apache.commons.logging.Log，通过该类的成员方法，我们就可以将不同性质的日志信息输出到目的地（目的地由配置文件指定，可能是stdout，也可能是文件，还可能是发送到邮件，甚至发送短信到手机……）

13.4.4. 日志的配置方法

如果在pkuas中需要更灵活地对日志服务进行一些自定义配置，则可以通过修改pkuas.xml文件中的日志配置部分来实现。下面讲讲几种主要的配置方法。

1、自定义logger

通过建立一个单独的logger，来输出需要的特定信息。如下例子：

首先定义appender，指明输出方式、级别、格式等：

```
<!--
added by zhanglei09
add an appender named "myTestAppender"
for special usage.
-->
            <appender name="myTestAppender" class="org.apache.log4j.RollingFi
<param name="File" value="../logs/myTest.log"/>
                    <param name="Append" value="false"/>
                    <!-- set Level -->
              <param name="Threshold" value="debug"/>
              <param name="MaxFileSize" value="300KB"/>
                    <param name="MaxBackupIndex" value="10"/>

             <layout class="org.apache.log4j.PatternLayout">
                       <param name="ConversionPattern"
                        value="%d{HH:mm:ss} %l [%t]%c(%p) %m%n"/>
             </layout>
            </appender>
```

接着定义一个单独的logger，如下：

```
<!--
               added by zhanglei09
            add a logger named "myTestLog"
            to use it, org.apache.commons.logging.Log log = org.apache.commor
            -->
<logger name="myTestLog" additivity="false">
       <level class="org.apache.log4j.Level" value="debug"/>
       <appender-ref ref="myTestAppender"/>
   </logger>
```

这样，便能在代码中需要使用该logger的地方引用这个logger了。获得logger的方法如下：

```
private static Log logger = LogFactory.getLog( “myTestLog” );
```

2、往远程主机写日志

可以使用socket appender。如下示例：

```
<appender name="socketAppender" class="org.apache.log4j.RollingFileAppender">
 <param name="RemoteHost" value="localhost"/>
    <param name="Port" value="5001"/>
   <param name="LocationInfo" value="true"/>
    <layout class="org.apache.log4j.PatternLayout">
          <param name="ConversionPattern"
               value="[start]%d{DATE}[DATE]%n%p[PRIORITY]%n%x
       [NDC]%n%t[THREAD]%n%c[CATEGORY]%n%m[MESSAGE]%n%n"/>
 </layout>
</appender>
```

3、使用自定义的格式

可以通过继承Log4j的Layout类及Parser类，来自定义新的日志布局器和解析器。

如，为了定位到bundle信息，定义了Log4j的Layout类及Parser类的子类，为UserBundlePatternLayout及UserBundlePatternParser，放在pku.as.logger包下。这两个类可以针对以下特定的格式输出bundle信息。在定义格式时，用%.n#可以输出所在bundle名称，其中，n为一整数，用于容纳包名，再根据包名映射为bundle名。一般n设为50就合适。

例：

```
<layout class="pku.as.logger.UserBundlePatternLayout">
 <param name="ConversionPattern"
          value="%d{HH:mm:ss} [BUNDLE:%.50#] - %m%n"/>
</layout>
```

4、建议使用的模板

下面以建立控制台输出的appender为例，给出一个建议的模板。如下：

```
<appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
       <param name="Threshold" value="info"/>
 <layout class="pku.as.logger.UserBundlePatternLayout">
     <param name="ConversionPattern"
            value="%d{HH:mm:ss} %5p [THREAD:%t] (LOCATION:%F:%4L)
 </layout>
</appender>
```

控制台输出示例：

```
11:57:53  INFO [THREAD:FelixStartLevel] (LOCATION:Main.java:  84) [BUNDLE:SERVICE_
11:57:53  INFO [THREAD:FelixStartLevel] (LOCATION:Main.java: 377) [BUNDLE:SERVICE_
```

说明：输出的每一行信息依次为：时间、日志级别、所在线程、位置（文件以及文件中的行数）、所在bundle以及相应的具体日志信息。

13.4.5. 日志内容示例

下面是以“日志服务的配置示例”部分的示例文件为配置文件，PKUAS-2010中将生成的实际日志内容。

启动服务器后，控制台会输出相应日志内容，另外，根目录下的log文件夹中，pkuas.log及pkuas-web.log中也有相应的日志输出。

下面是控制台输出内容：

```
//pku.as.launcher.felix.PKUAS.java中的System.out.println()
Welcome to OW2 PKUAS (Running on Felix).
   Type 'help' for available commands.
-----------------------------------------------

//pku.as.sm.Main.java中的System.out.println()
-> PKUAS V2008 initializing...

//以下为调用日志语句
//private static Log logger = LogFactory.getLog(Main.class);
//……
//logger.info("Initialization phase complete.");
//后的输出内容
11:55:29[FelixStartLevel]Main(INFO) Initialization phase complete.
```

```
11:55:29[FelixStartLevel]Main(INFO) Registering kernel MBeans...
11:55:29[FelixStartLevel]Main(INFO) Starting CodeBase service...

//Pku.as.codebase.CodeBaseService.java中调用日志
11:55:29[FelixStartLevel]CodeBaseService(INFO) Codebase : http://192.168.1.219:807
11:55:29[FelixStartLevel]CodeBaseService(INFO) Woker threads:2
//Pku.as.codebase.CodeBaseService.java.

11:55:29[FelixStartLevel]Main(INFO) CodeBase service started.
11:55:29[FelixStartLevel]Main(INFO) Starting the Service Manager...
11:55:30[FelixStartLevel]Main(INFO) Service Manager started.
11:55:30[FelixStartLevel]Main(INFO) Starting Application Manager...
11:55:30[FelixStartLevel]Main(INFO) Application Manager started.
11:55:30[FelixStartLevel]Main(INFO) Starting JSR77 Manager...
11:55:30[FelixStartLevel]Main(INFO) JSR77 Manager started.
11:55:30[FelixStartLevel]Main(INFO) PKUAS V2008 started!
//pku.as.sm.Main.java.

……
11:55:47[Configuration Updater]PoolManager(INFO) Execute SQL script file (TablePop
11:55:47[ContainerBackgroundProcessor[StandardEngine[PKUASWebContainer]]]WebappLoa

->
->
//输入停止服务器后的日志输出
-> shutdown
-> 11:58:05[FelixStartLevel]ApplicationManager(INFO) Auto-deploy thread stopping..
11:58:05[FelixStartLevel]ApplicationManager(INFO) Application Manager stopped!
11:58:05[FelixStartLevel]Server(INFO) Stopping WebContainer(tomcat5)...
11:58:05[FelixStartLevel]Http11BaseProtocol(INFO) Stopping Coyote HTTP/1.1 on http
11:58:07[FelixStartLevel]MessageServices(INFO) Try to stop all message services...
11:58:07[FelixStartLevel]MessageServices(INFO) All the message services have been
11:58:07[FelixStartLevel]JORAMAdapter(INFO) Try to stop JORAM...
11:58:08[FelixStartLevel]JORAMAdapter(INFO) JORAM stopped!
11:58:08[FelixStartLevel]Main(INFO) Shutdown command received.
11:58:08[FelixStartLevel]Main(INFO) PKUAS V2008 shutting down...
11:58:08[FelixStartLevel]Main(INFO) Shutting down applications...
11:58:08[FelixStartLevel]Main(INFO) Shutting down services...
PKUAS V2008 halted.
}}}
```

下面是Pkuas.log记录内容

```
2010-04-19 11:55:29,609 [FelixStartLevel] INFO pku.as.sm.Main - Initialization ph
```

```
2010-04-19 11:55:29,609 [FelixStartLevel] INFO pku.as.sm.Main - Registering kerne
2010-04-19 11:55:29,609 [FelixStartLevel] INFO pku.as.sm.Main - Starting CodeBase
2010-04-19 11:55:29,609 [FelixStartLevel] INFO pku.as.codebase.CodeBaseService -
2010-04-19 11:55:29,609 [FelixStartLevel] INFO pku.as.codebase.CodeBaseService -
2010-04-19 11:55:29,718 [FelixStartLevel] INFO pku.as.sm.Main - CodeBase service
2010-04-19 11:55:29,718 [FelixStartLevel] INFO pku.as.sm.Main - Starting the Serv
2010-04-19 11:55:30,546 [FelixStartLevel] INFO pku.as.sm.Main - Service Manager s
2010-04-19 11:55:30,546 [FelixStartLevel] INFO pku.as.sm.Main - Starting Applicat
2010-04-19 11:55:30,546 [FelixStartLevel] INFO pku.as.sm.Main - Application Manag
2010-04-19 11:55:30,546 [FelixStartLevel] INFO pku.as.sm.Main - Starting JSR77 Ma
2010-04-19 11:55:30,546 [FelixStartLevel] INFO pku.as.sm.Main - JSR77 Manager sta
2010-04-19 11:55:30,546 [FelixStartLevel] INFO pku.as.sm.Main - PKUAS V2008 start
……
```

下面是Pkuas-web.log记录内容

```
2010-04-19 11:55:37,312 [Configuration Updater] INFO org.apache.coyote.http11.Htt
2010-04-19 11:55:37,343 [Configuration Updater] INFO org.apache.catalina.core.Sta
2010-04-19 11:55:37,343 [Configuration Updater] INFO org.apache.catalina.core.Sta
2010-04-19 11:55:37,375 [Configuration Updater] INFO org.apache.coyote.http11.Htt
……
```

可以看出，在pku包下的日志内容记录在了pkuas.log里，而在apache包下的日志内容记录在了pkuas-web.log里。控制台则输出了所有日志信息。

第 14 章 PKUAS应用开发指南

14.1. 引言

本开发指南面向使用PKUAS来设计、开发应用系统的人员，目的是指导有关人员开发、组装和部署基于PKUAS的应用系统。

阅读本开发指南需要首先了解和掌握J2EE/EJB应用的开发方法，本指南对EJB、JSP/Servlet等实际开发不做具体介绍，这方面内容请参看SUN公司有关文档和其它参看书籍。

本开发指南将按照一般的开发流程，从准备工作开始，逐步介绍EJB、WEB module的开发、应用系统的组装和部署，之后对高级特性的使用进行介绍。

14.2. 应用服务器知识概述

14.2.1. 什么是应用服务器

一般的，软件可以被划分为三类：系统软件、支撑软件和应用软件。

其中，系统软件主要是指操作系统等与系统资源密切相关的软件，常见的如MS Window系列和Unix系列OS，都可以被看作是系统软件；支撑软件是指软件开发工具等支持开发过程中的软件，常见的IDE就属于这种软件；这两种软件可以合称为基础软件。应用软件指特定于应用领域的专用软件。

我们所要探讨的应用服务器是一种为方便应用软件开发与维护而逐步形成的基础软件：应用服务器是网络环境中应用系统的高层运行平台；它使应用系统的代码更为简单，使开发人员的精力可以更加集中于系统的逻辑部分。（《应用服务器原理与实现》）

14.2.2. 应用服务器的出现环境

网络软件（分布在网络环境中的软件系统）不同于单机环境中的软件：它要求分布在网络中的各个节点能够相互协作从而共同完成目标任务，因此网络软件必需解决网络环境里分布性、可靠性、安全性等问题。解决这些问题的一条策略就是：提取软件共性成分，屏蔽系统低层的复杂度，从而在高层保持复杂度的相对稳定。采用了这一策略的典型就是操作系统，数据库管理系统，和应用服务器。

根据前述理由可以知道，一个具体的应用系统，如网络软件，直接从操作系统层开始开发是不合理的，而应该以一个相对高层的平台为起点。这个平台为应用系统的开发、维护提供基础。应用服务器将网络软件的共性成分从应用系统中剥离出来，构成相对完整、独立的系统软件；因其屏蔽了低层网络细节，是开发者能够将精力集中于目标系统的业务逻辑上，从而简化应用系统的开发和维护的过程。是不是很像操作系统？操作系统屏蔽了低层硬件细节，为计算机使用者提供的是一系列应用接口，展示了计算机在逻辑层面的应用。

纵向来看，应用服务器位于操作系统之上，应用系统之下；横向来看，应用服务器位于软件的表示层和数据层之间的业务逻辑层。

14.2.3. 应用服务器的功能

具体来说，应用服务器有以下功能：

- 通过构件容器为构建提供基本的运行环境，包括管理构件生命周期、管理构件的实例、管理构建的信息等；
- 提供高层通信服务，以满足网络环境中的构件之间的互操作，包括业务层与表示层之间的通信、业务层与数据层之间的通信、业务层内部公共服务与应用层之间的通信；
- 提供公共服务，包括查找服务、事务服务和安全服务。

14.3. 准备工作

开发者在开发J2EE应用前需要作相关的准备工作。包括JDK、应用服务器和数据库的安装、设置等。J2SE和J2EE的API文档可以给开发者提供很多帮助。此外，开发者可能还需要选择一个合适的集成开发环境来提高工作效率。

14.3.1. 系统需求

PKUAS运行系统的最低配置为：奔腾166MHZ或同等性能机器，带有48兆以上内存。安装Java 2 SDK需要至少70M的硬盘空间，推荐使用200M以上的空闲磁盘空间。

PKUAS可以在装有Microsoft Windows 95,98(1st or 2nd Edition), NT4.0 with Service Pack 5,ME,2000 Professional,2000 Server,2000 Advanced Server, XP Operating System和类Unix环境(Linux,Solaris)的机器上运行。

运行PKUAS必须预先安装Java 2 SDK 1.4以上版本并配置环境变量，并对PKUAS进行正确配置。

14.3.2. 数据源的配置

PKUAS支持MySQL、Oracle、SQL Server、Cloudscape等多个数据库的使用。实际上所有JDBC支持的数据库在PKUAS中都可以使用。

以MySQL为例，在安装好MySQL后，可以在pkuas.xml文件中加入以下小节来配置数据源：

```
<!-- Datasource (JNDI name: jdbc/petstore) for our petstore application -->

<Service Class="pku.as.datasvc.PoolDataSource"   Name="jdbc/petstore">

        <Attribute Name="expirationTime" Value="300" />

        <Attribute Name="minCapacity" Value="10" />

        <Attribute Name="user" Value="hgj61" />

        <Attribute Name="password" Value="hgj61" />

        <Attribute Name="URL"  Value="jdbc:mysql://192.168.4.137:3306/petsto

        <Attribute Name="driverClassName"  Value="com.mysql.jdbc.Driver" />

        <Attribute Name="maxCapacity" Value="1000" />

</Service>
```

其中Class="pku.as.datasvc.PoolDataSource" Name=" jdbc/petstore"是这个数据源的名字， jdbc/petstore是开发者在开发J2EE应用时要用到的数据源字符串，根据需要可以任意取名。用户名、密码、URL等属性值可根据使用者的情况填写。

"URL"属性的值供JDBC建立连接时用，开发者只需要将MysqlServerAddress换成实际的mysql所处的机器地址，后面的EJBTEST是准备用的数据库名。这样数据源就配置好了。

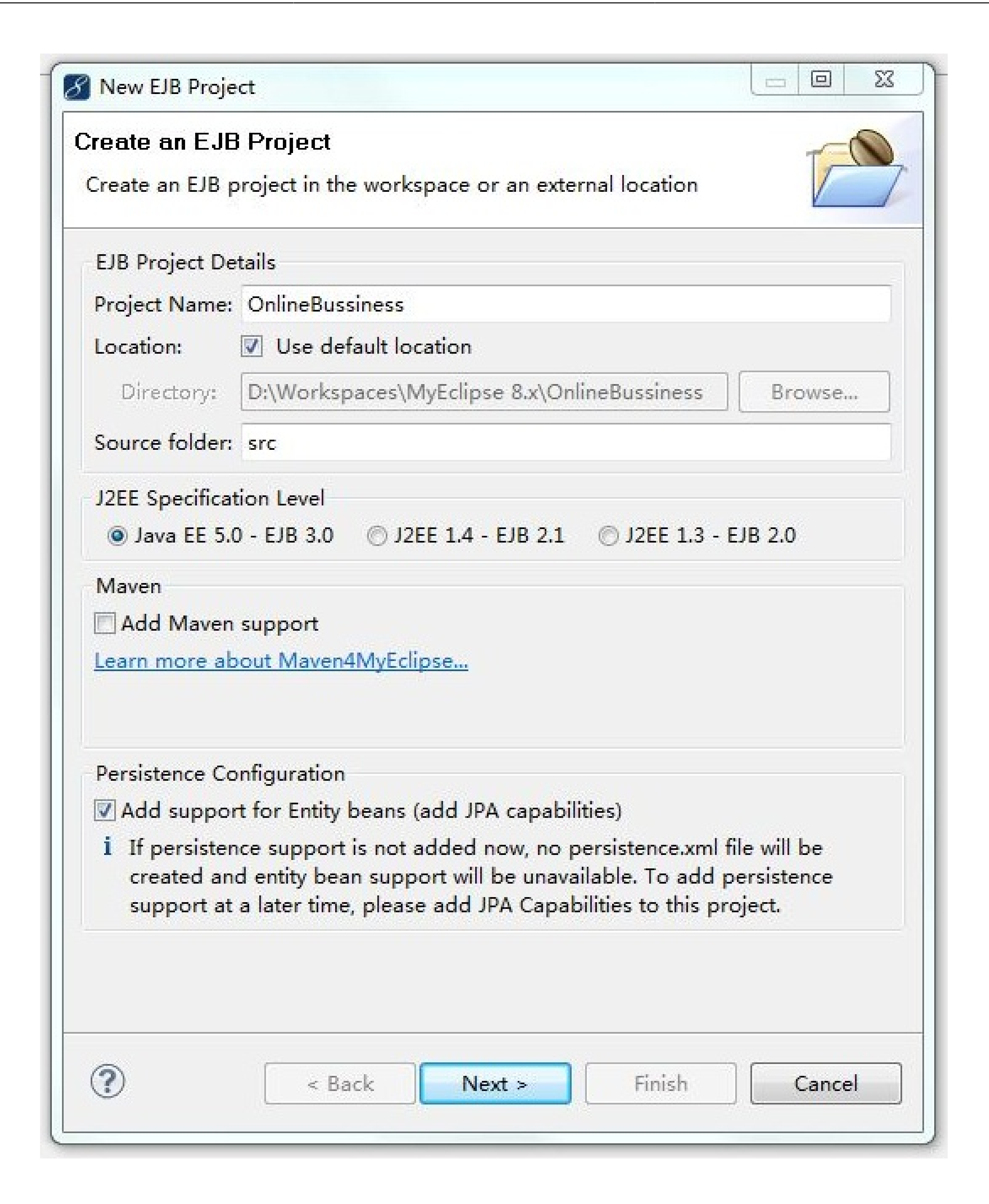
14.4. EJB模块的开发

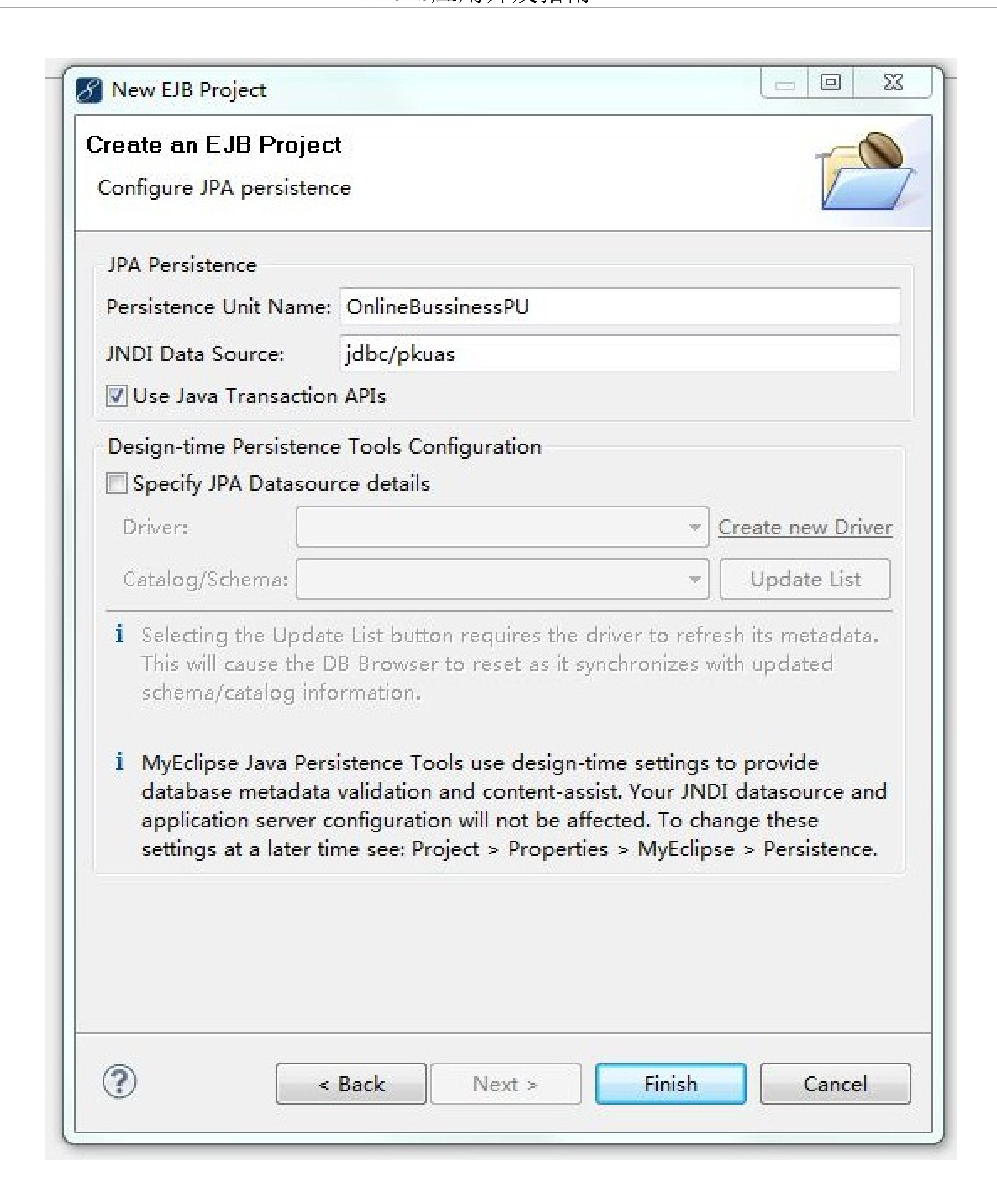
EJB可以分为三种：Session Bean（会话Bean），Entity Bean（实体Bean），和Message-Driven Bean（消息驱动Bean）。

本手册将以一个网上购物的应用为例，展示在pkuas中开发这几种Bean的方法。开发环境是MyEclipse8.0GA。

14.4.1. 创建Project

File->New->EJB Project


New EJB Project
Create an EJB Project
Create an EJB project in the workspace or an external location
EJB Project Details
Project Name: OnlineBussiness
Location: Use default location
Directory: D:\Workspaces\MyEclipse 8.x\OnlineBussiness
Browse...
Source folder: src
J2EE Specification Level
Java EE 5.0 - EJB 3.0
J2EE 1.4 - EJB 2.1
J2EE 1.3 - EJB 2.0
Maven
Add Maven support
Learn more about Maven4MyEclipse...
Persistence Configuration
Add support for Entity beans (add JPA capabilities)
If persistence support is not added now, no persistence.xml file will be created and entity bean support will be unavailable. To add persistence support at a later time, please add JPA Capabilities to this project.
< Back
Next >
Finish
Cancel



Finish

14.4.2. 开发无状态会话Bean

14.4.2.1. 概述

无状态会话Bean是具有类级别注解@Stateless的任意标准Java类。它由业务接口和业务方法两部分构成。

为了演示无状态会话Bean的使用，我们将创建SearchFacade会话Bean，它为客户提供关于现有的酒类的搜索。

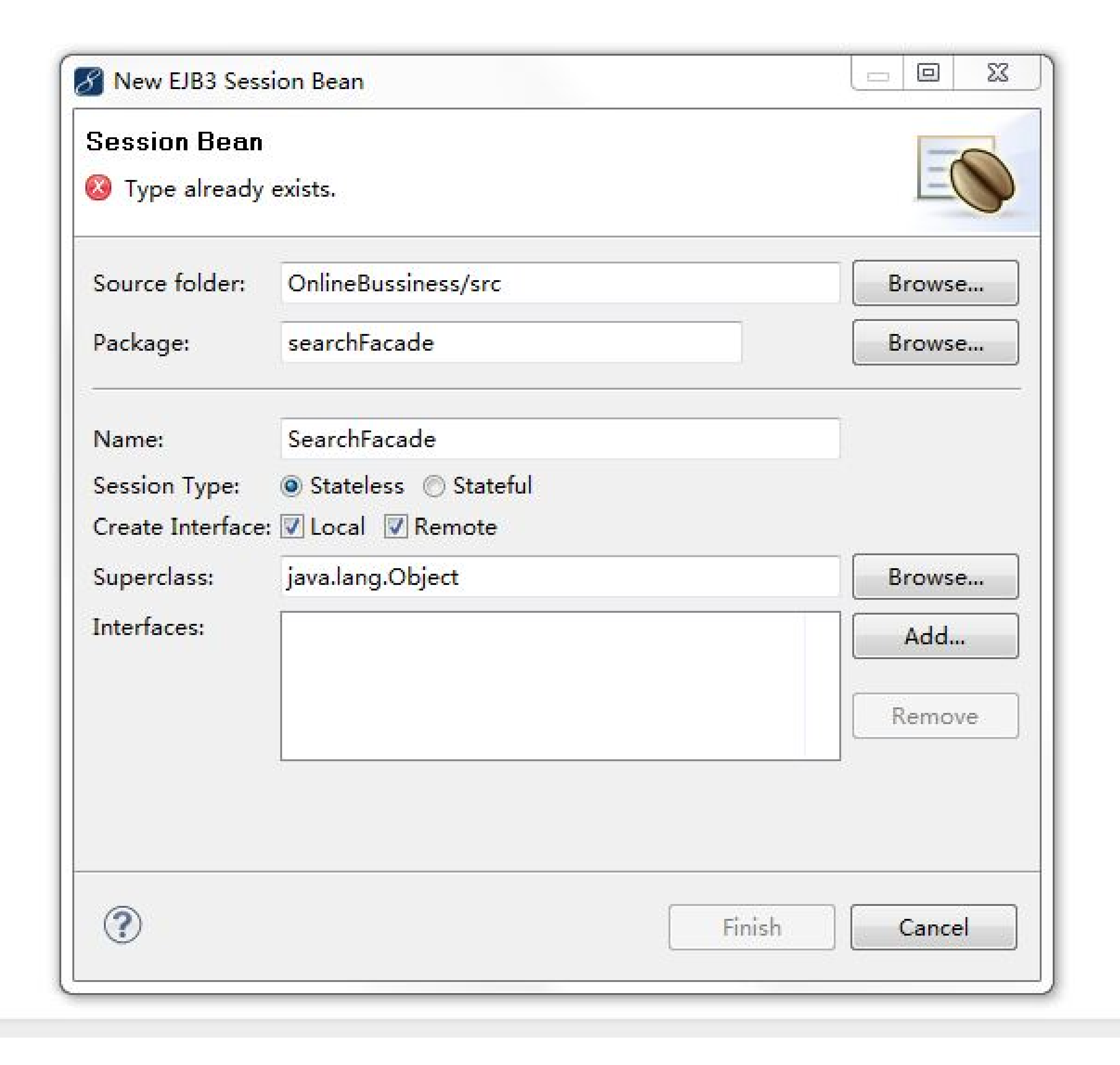
14.4.2.2. 开发流程

无状态会话Bean的业务接口分为：

- 本地业务接口——@Local，默认，用于客户端和EJB容器在同一个JVM上的情况
- 远程业务接口——@Remote，用于客户端和EJB容器不在同一个JVM上的情况

下面演示具体的步骤。

14.4.2.2.1. 创建会话Bean：File->New->EJB3 SessionBean



14.4.2.2.2. 编写业务接口

本地业务接口：SearchFacadeLocal.java

```
package searchFacade;

import java.util.List;

import javax.ejb.Local;

@Local

public interface SearchFacadeLocal {

        List<String> wineSearch(String wineType);

}
```

远程业务接口：SearchFacadeRemote.java

```
package searchFacade;

import java.util.List;

import javax.ejb.Remote;

@Remote

public interface SearchFacadeRemote {

        List<String> wineSearch(String wineType);

}
```

14.4.2.2.3. 编写业务方法

SearchFacade.java

```
package searchFacade;

import java.util.ArrayList;

import java.util.List;

import javax.ejb.Stateless;

@Stateless(name="SearchFacade")

public class SearchFacade implements SearchFacadeLocal, SearchFacadeRemote, java.io.Seriali

        public SearchFacade(){}

        public List<String> wineSearch(String wineType) {

                List<String> wineList = new ArrayList<String>();

                if(wineType.equals("Red")) {

                        wineList.add("Bordeaux");

                        wineList.add("Merlot");

                        wineList.add("Pinot Noir");

                }

                else if (wineType.equals("White")) {

                        wineList.add("Chardonnay");

                }

                return wineList;

        }

}
```

14.4.2.2.4. 配置jndi

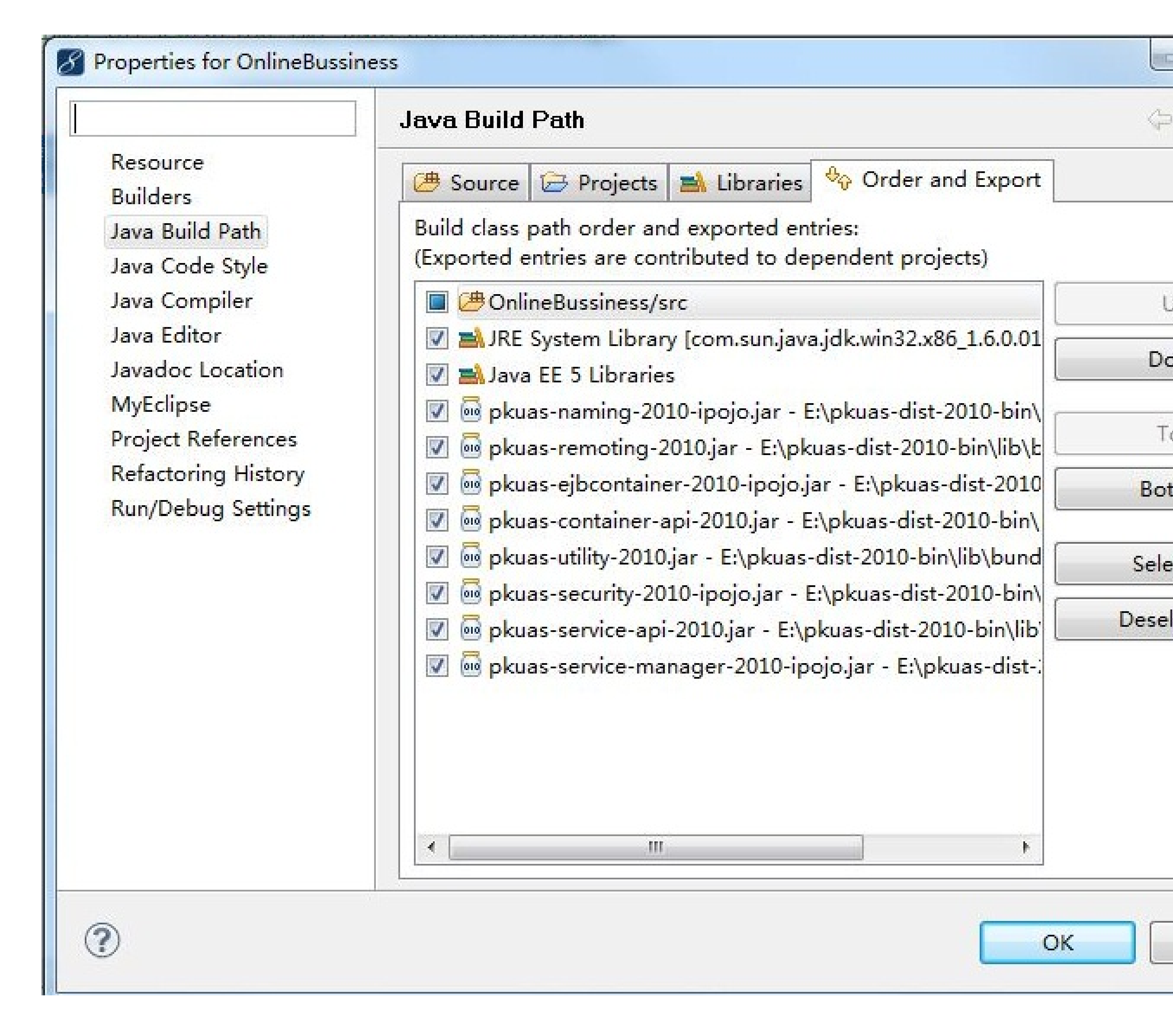
```
java.naming.factory.initial=pku.as.naming.SmartCtxFactory

java.naming.provider.url=rmi\://localhost\:2001
```

这个文件要放在工程的根目录下。

14.4.2.2.5. 配置Build Path

工程名右键->Build Path->Configure Build Path



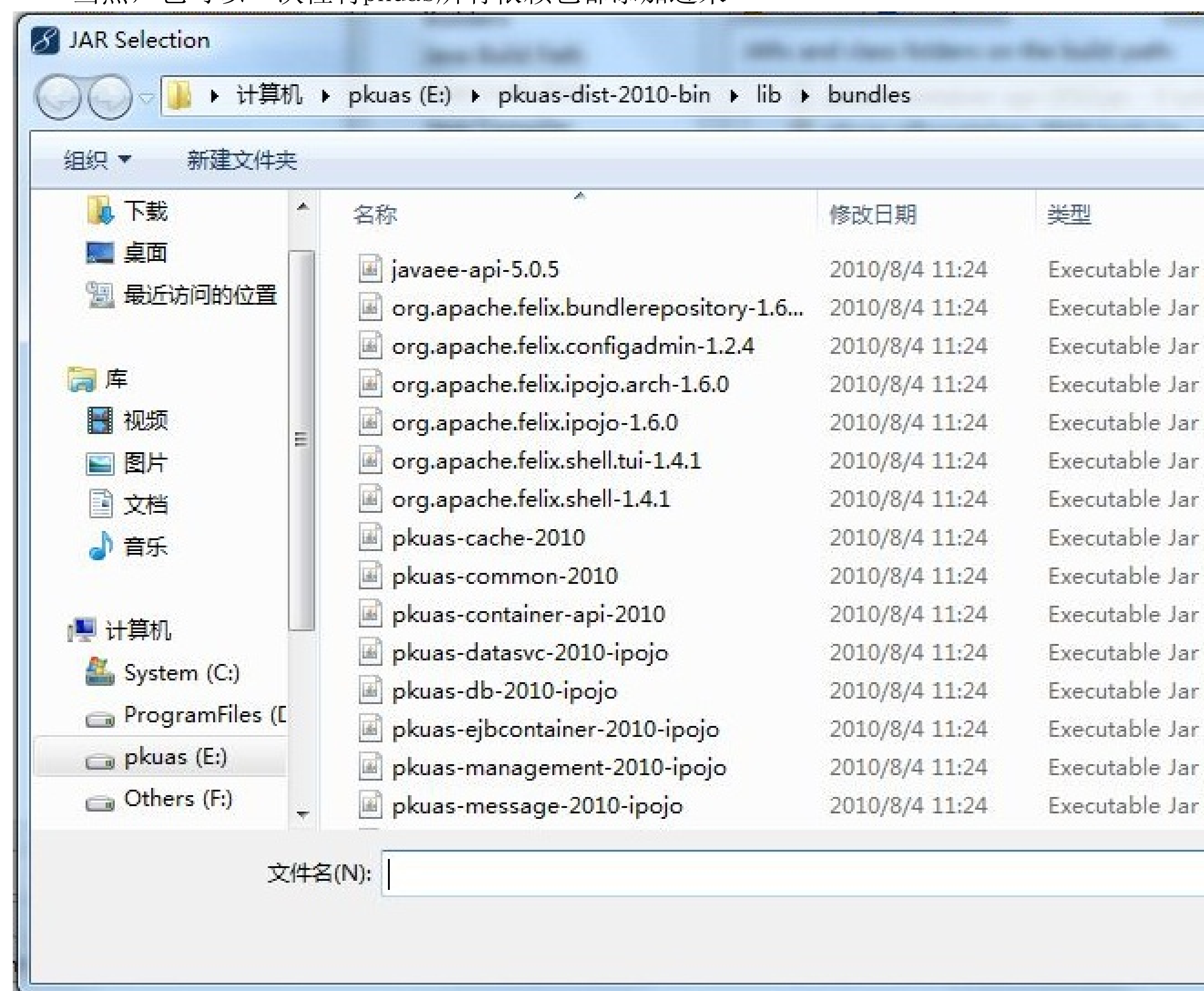
Libraries->Add External JARs

在pkuas的lib目录下选择依赖包，本例所依赖的包是

- pkuas-naming-2010-ipojo.jar
- pkuas-remoting-2010.jar
- pkuas-ejbcontainer-2010-ipojo
- pkuas-container-api-2010.jar
- pkuas-utility-2010.jar
- pkuas-security-2010-ipojo.jar
- pkuas-service-api-2010.jar
- pkuas-serviece-manager-2010-ipojo.jar

将这些包添加进来后，再在Order and Export中选中添加即可。

当然，也可以一次性将pkuas所有依赖包都添加进来



14.4.2.2.6. 启动PKUAS

pkuas目录下bin目录下启动run.bat.

控制台显示"Application Manager is started!"信息表示pkuas启动成功。

14.4.2.2.7. 打包部署

将SearchFacade包打包成JAR，并将这个JAR包放在pkuas目录下的application目录里。

这是pkuas控制台提示"###The app [SearchFacade.jar] deployed###"，表示这个EJB已经成功部署。

14.4.2.2.8. 在客户端测试运行EJB

为了成功演示EJB的执行结果，还要编写一个客户端来调用这个EJB。在本工程下创建另一个包Client，在包里添加如下java文件：

SearchFacadeClient.java

```
package client;

import searchFacade.*;

import java.util.List;

import javax.naming.*;

public class SearchFacadeClient {

       public SearchFacadeClient(){}

       public static void main(String[] args) throws NamingException {

              Context ctx = new InitialContext();

              SearchFacadeRemote response = (SearchFacadeRemote)ctx.lookup("SearchFacade"

              System.out.println("SearchFacade Lookup");

              System.out.println("Searching wines");

              List<String> winesList = response.wineSearch("Red");

              System.out.println("Printing wines list");

              for(String wine:(List<String>)winesList) {

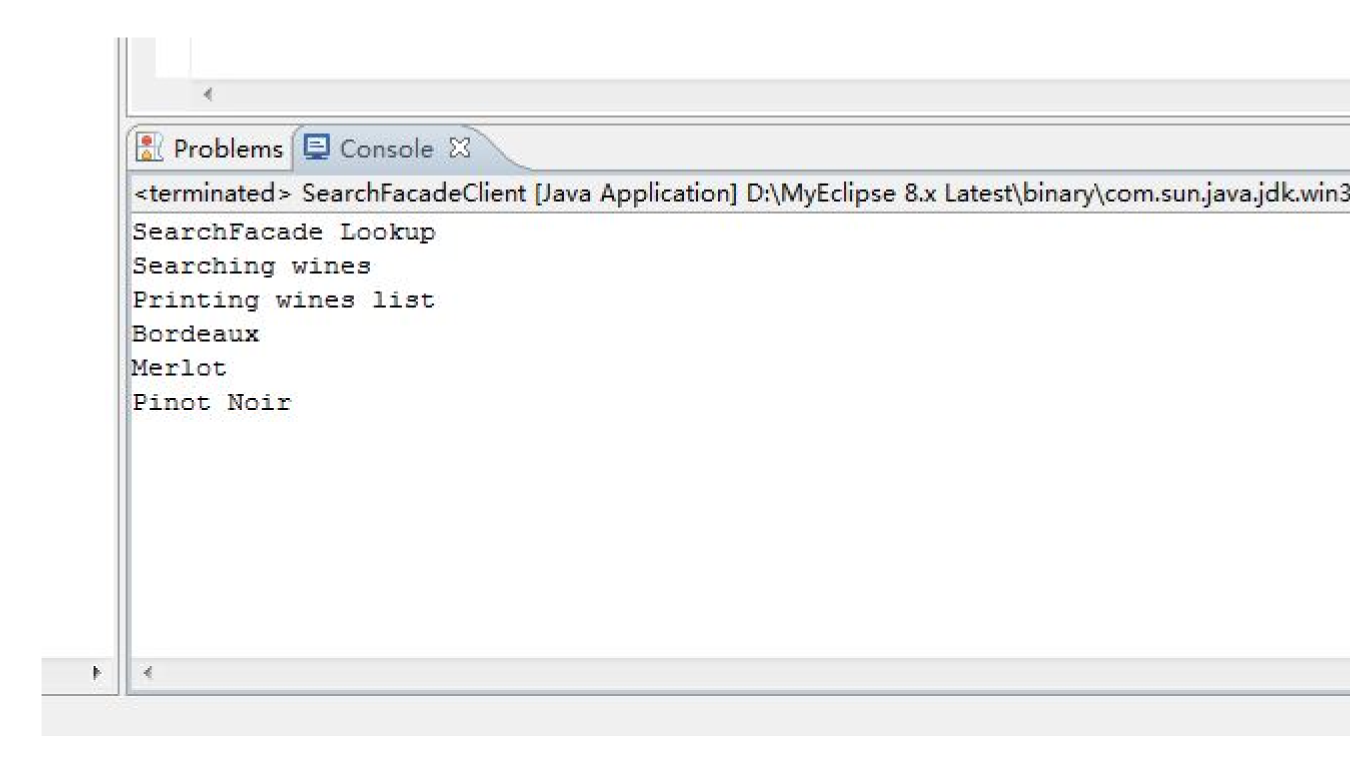
                     System.out.println(wine);

              }

       }

}
```

运行这个客户端程序，控制台显示如下信息，表示我们编写的会话Bean已经成功被调用了：



14.4.3. 开发有状态会话Bean

14.4.3.1. 概述

有状态会话Bean是具有类级别注解@Stateful的任意标准Java类。和有状态会话Bean一样，它也由业务接口和业务方法两部分构成。

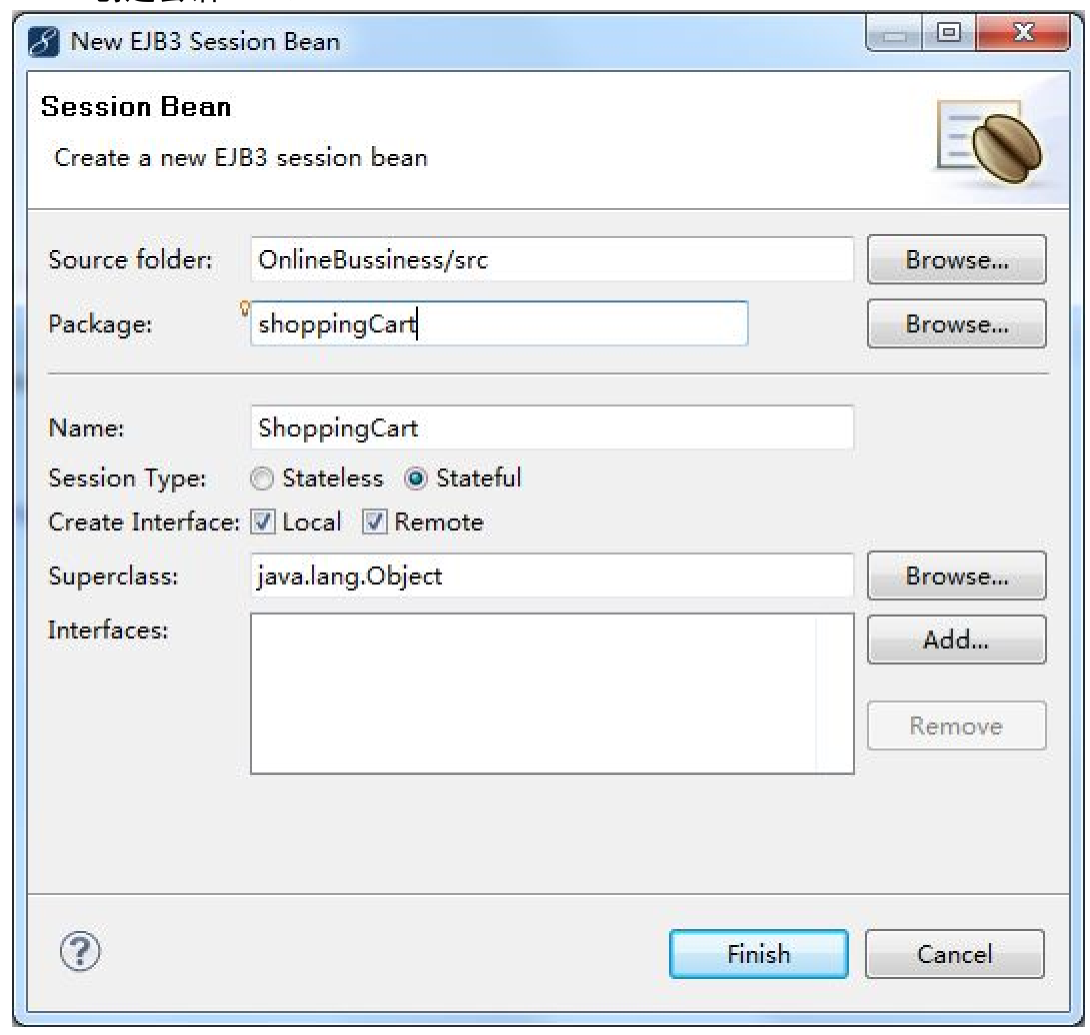
每个有状态会话Bean在bean实例的生命周期内都只服务于一个客户端；它就像是客户端的代理。不同与无状态会话Bean，有状态会话Bean的实例不会在EJB对象之间进行切换，也不会被置于实例池内。一旦有状态会话Bean被实例化，并被指派给了某个EJB对象，他在整个生命周期内只会关联于该EJB对象。

为了演示有状态会话Bean的使用，我们将创建ShoppinCart会话Bean，它跟踪添加到用户购物车的项目以及它们各自的数量。

14.4.3.2. 开发流程

下面演示具体的步骤。

14.4.3.2.1. 创建会话Bean：File->New->EJB3 SessionBean



14.4.3.2.2. 编写业务接口

本地业务接口：ShoppingCartLocal.java

```
package shoppingCart;

import java.util.ArrayList;

import javax.ejb.Local;

@Local
public interface ShoppingCartLocal {

        public void addWineItem(String wine);

        public void removeWineItem(String wine);

        public void setCartItems(ArrayList<String> cartItems);

        public ArrayList<String> getCartItems();

}
```

远程业务接口：ShoppingCartRemote.java

```
package shoppingCart;

import java.util.ArrayList;

import javax.ejb.Remote;

@Remote
public interface ShoppingCartLocal {

        public void addWineItem(String wine);

        public void removeWineItem(String wine);

        public void setCartItems(ArrayList<String> cartItems);

        public ArrayList<String> getCartItems();

}
```

14.4.3.2.3. 编写业务方法

ShoppingCart.java

```
package shoppingCart;

import javax.annotation.*;
import javax.ejb.Remove;
import javax.ejb.Stateful;

import java.util.ArrayList;

@Stateful(name="ShoppingCart")
public class ShoppingCart implements ShoppingCartLocal, ShoppingCartRemote, java.io.Serializable {

	public ShoppingCart(){};

	public ArrayList<String> cartItems;

	@PostConstruct
	public void initialize() {
		cartItems = new ArrayList<String>();
	}

	@PreDestroy
	public void exit() {
		// items list into the database
		System.out.println("Saved items list to database");
	}

	@Remove
	public void stopSession() {
		// the method body can be empty
		System.out.println("From stopSession method with @Remove annotation");
	}

	public void addWineItem(String wine) {
		cartItems.add(wine);
	}

	public void removeWineItem(String wine) {
		cartItems.remove(wine);
	}

	public void setCartItems(ArrayList<String> cartItems) {
		this.cartItems = cartItems;
	}

	public ArrayList getCartItems() {
		return cartItems;
	}
}
```

14.4.3.2.4. 配置jndi

同无状态会话Bean

14.4.3.2.5. 配置Build Path

同无状态会话Bean

14.4.3.2.6. 启动pkuas

同无状态会话Bean

14.4.3.2.7. 打包部署

将ShoppingCart包打包成JAR，并将这个JAR包放在pkuas目录下的application目录里。

这是pkuas控制台提示"###The app [ShoppingCart.jar] deployed###"，表示这个EJB已经成功部署。

14.4.3.2.8. 在客户端测试运行EJB

为了成功演示EJB的执行结果，还要编写一个客户端来调用这个EJB。在本工程下创建另一个包Client，在包里添加如下java文件：

ShoppingCartClient.java

```
package client;

import shoppingCart.*;

import java.util.*;

import javax.naming.*;

public class ShoppingCartClient {

        public ShoppingCartClient() {}

        public static void main(String[] args) throws NamingException{

                Context ctx = new InitialContext();

                ShoppingCartRemote response = (ShoppingCartRemote)ctx.lookup("ShoppingCart")

                System.out.println("ShoppingCart Lookup");

                System.out.println("Adding Wine Item");

                response.addWineItem("Zinfandel");

                System.out.println("Printing Cart Items");

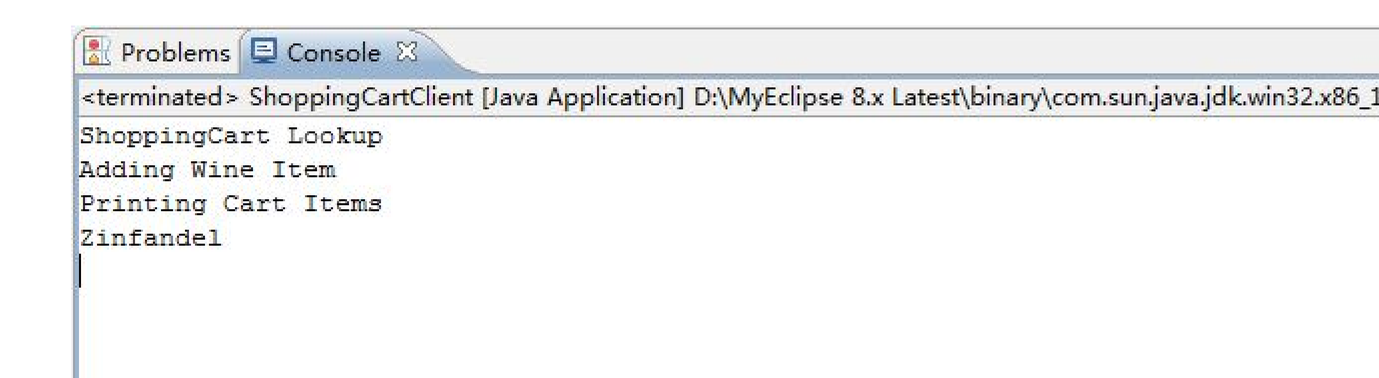
                ArrayList<String> cartItems = response.getCartItems();

                for (String wine : (List<String>) cartItems) {

                        System.out.println(wine);
                }
        }

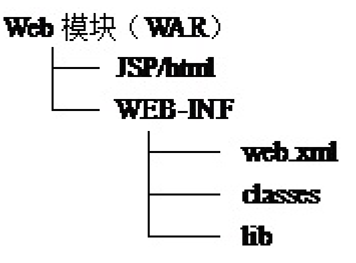
}
```

运行这个客户端程序，控制台显示如下信息，表示我们编写的会话Bean已经成功被调用了：



14.5. WEB模块的开发

PKUAS集成了Tomcat（目前集成的版本是5.5）作为Servlet引擎。Web模块的打包结构与在Tomcat上面单独部署基本上是一样的。应用的结构为：



JSP/html表示这个Web中用到的静态页面和JSP页面；WEB-INF目录下面的web.xml是Web应用的描述文件。classes目录下面按包目录放了这个Web模块使用的JavaBeans和Servlet的类。而lib目录下面放了Web应用中使用到的JAR包。

WEB模块可以直接在PKUAS上部署，或作为一个企业应用（Enterprise Application，以EAR的形式部署到PKUAS上）的组成部分：

作为独立的WEB应用部署:

- 直接将WEB 模块的目录放到PKUAS下面的application目录下
- 将WEB 模块打包为WAR文件后将WAR放到application目录下面

这两种方式部署的WEB 模块在启动PKUAS后可以直接访问。

作为一个企业应用的一部分:

- 将WEB 模块打包为WAR文件，和整个应用一起部署。在下一章中，将介绍组装和部署的过程。
- 将WEB 模块打包为WAR的过程与上面打包EJB相似，也是使用jar命令，这时后面的参数中，打包文件的扩展名为war。

用户除了使用jar命令外，还可以使用maven、ant等工具来自动实现编译和打包，另外Myeclipse以及安装了web开发插件的eclipse也可以将web项目导出为war包或ear包。如果用户都没有安装这些的话，还可以通过解压工具（如WinRAR,7zip等）将web工程目录压缩为zip格式，同时将后缀名改为war或ear，需要注意的如果

选择这种方式，对于web项目所需要的web.xml和ear项目所需要的application.xml需要注意审查，因为手动打包不会检查这些文件的正确性.

对独立部署的Web 模块，如果它还需要调用其它EJB，则需要将这些EJB的接口定义以及PKUAS为其生成的Stub/Skeleton都放到classes目录（以class文件的形式）或者lib目录（以JAR包形式）下面。

14.6. 应用的组装和部署

应用的组装就是将EJB 模块和WEB 模块进行组合并配置为一个完整应用系统的过程；应用的部署是将组装得到的系统放入特定的运行环境、并根据实际环境进行必要的配置，使系统可以运行的过程。从本质上讲，组装和部署应该是两个独立的过程。但在实际开发过程中，这两个过程的界限很难划分，通常是同时进行的。因此，本文将两个过程合并一同说明。

另外，当直接在PKUAS上部署EJB模块时，可以将这个EJB模块看成只包含一个EJB模块的应用。因此，在下面的把EJB模块看成特殊的应用，与一般企业应用一同介绍。

14.6.1. 对Web模块的描述文件作必要的修改

在组装和部署应用的时候，也需要对Web模块的描述文件进行修改和增加。在PKUAS中，对Web模块的描述文件主要为J2EE和JSP/Servlet规范定义的web.xml；另外，在web.xml中也会有一些关于Web模块的特定于PKUAS的信息，这部分将在后面进行介绍。

14.6.1.1. web.xml

web.xml中的绝大多数内容都在开发Web模块的时候写好，这里只需要对其中的信息做一点补充。

在web.xml中，用ejb-ref元素来描述Web module中要使用的EJB信息，例如：

```
<ejb-ref>

    <ejb-ref-name>ejb/cart/Cart</ejb-ref-name>

    <ejb-ref-type>Session</ejb-ref-type>

    <home>com.sun.j2ee.blueprints.shoppingcart.cart.ejb.ShoppingCartHome</home

    <remote>com.sun.j2ee.blueprints.shoppingcart.cart.ejb.ShoppingCart</remote

    <ejb-link>CartEJB</ejb-link>

  </ejb-ref>
```

这里描述了一个在程序中使用的名字为CartEJB的EJB，其EJB类型为会话（Session）EJB，Home和Remote接口分别是

com.sun.j2ee.blueprints.shoppingcart.cart.ejb.ShoppingCartHome和
com.sun.j2ee.blueprints.shoppingcart.cart.ejb.ShoppingCart。

引用的EJB名字为CartEJB。在实际的开发过程中，被引用的EJB可能在另一个存档文件中；一般地，用jar包路径接EJB名字的格式给出，中间以#标示。

如：

```
<ejb-link>EmployeeRecord</ejb-link>

<ejb-link>../products/product.jar#ProductEJB</ejb-link>
```

14.6.2. 编写描述文件

描述文件中保存了应用的配置信息。在PKUAS上面部署的应用，需要编写与应用相关的部署描述文件：

application.xml：由J2EE规范规定，对应用结构进行描述。

web.xml：特定于PKUAS的配置文件，对web应用的部署信息进行详细描述。

下面将分别介绍。

14.6.2.1. application.xml

application.xml主要描述应用本身及其包含的模块的静态信息。目前主要包括：

应用的信息：应用的名字display-name应用的描述descriptionEJB

模块信息：每个EJB模块中可以包含多个EJB，这里只描述EJB模块对应的物理文件（*.jar）。Web　模块信息：每个Web模块中包含一个可以通过Web方式访问的应用，这里描述Web对应的物理文件（*.war）及其启动后的路径（在URL中的信息）。

14.6.2.2. web.xml

web.xml位于每一个war包中，描述了部署在PKUAS上的web应用的一些具体信息，主要是外部资源引用。整个xml文件的信息结构如下：

- web：整个XML文件的最外层唯一元素，下面所有的元素都是它的子元素
    - resource-ref
        - 同pkuas-ejb.xml文件
    - resource-env-ref
        - 同pkuas-ejb.xml文件
    - login-config：供single-sign-on使用
        - auth-method
        - form-longin-config
            - form-login-page：声明用于给用户登录的页面
            - form-error-page：声明安全认证失败的页面

14.6.3. 打包应用

打包是指将前面开发的EJB、Web module以及组装、部署描述文件组合成一个可以在PKUAS中可以运行的文件。目前，PKUAS中支持在其之上部署EJB应用（只包含EJB，以JAR的形式部署）、Web应用（只包含Web页面、JSP/Servlet、Javabeans等）以及企业应用（包含EJB、Web module）。在本节中，将会分别介绍这三种应用的打包过程。

1．企业应用的打包过程：企业应用以EAR文件的形式部署到PKUAS上面。一个完整的EAR文件应该包括：

- 任意多个EJB的JAR
- 任意多个Web的WAR
- 组装、部署描述文件

2．EJB应用的打包过程：

打包后的应用就可以进行部署了。

EJB应用以JAR文件的形式部署到PKUAS上面。一个完整的JAR文件应该包括：

- 任意多个EJB：这个JAR为一个EJB模块，其中可以包含任意多个EJB，JAR中包含了EJB所需要的类文件。

3．Web应用的打包过程：与EAR应用的打包类似。

14.6.4. 应用的部署

在PKUAS中，部署即将打包好的应用放到PKUAS中应用部署目录中。具体的说，就是将EAR文件放到PKUAS安装目录下面的applications目录下。PKUAS会自动检测到应用的更新，如果是一个新的应用，那么PKUAS会直接部署，如果是一个已存在的应用的较新版本，那么PKUAS会将旧的应用卸载，然后部署新的应用。

还有一种方法就是通过网页端管理工具JSR77 Management Tools，在ear应用管理中，可以远程部署新的应用。

部署时，服务器为应用在cache文件夹中建立相应的目录，然后调用应用管理器对应用的各个部分进行解析和部署。

14.7. 应用的启动

目前，PKUAS中的应用启动与服务器启动同步，即启动PKUAS的同时，在部署目录（application）下面的所有应用都会被自动启动。

PKUAS提供自动部署功能，服务器会自动检测application目录，发现新的应用包或者应用包有所改动时，会重新部署该应用。

14.8. 高级特性

14.8.1. 使用事务

开发者在开发EJB构件的时候，可以利用应用服务器提供的事务服务来提高构件运行的可靠性。在构件开发过程中访问事务服务使得构件内部的操作具有ACID性质。根据J2EE规范，应用服务器给用户提供两种访问事务服务的途径：一种是编程式访问；另一种是声明式访问。

14.8.1.1. 声明式

Session Bean，Entity Bean和MessageDriven Bean都可以使用声明式事务服务。构件开发者只需要在应用的每一个构件的描述文件中声明事务服务的需求，不必关心服务器端具体的实现代码是怎么样的。例如，在eshop应用中有一个构件叫Order(Order是一个实体Bean)，开发者希望Order构件的create()方法（对应Bean类中的ejbCreate()方法。）在事务环境中执行，也就是希望在开始执行create()方法前必须已经启动事务，执行完create()后结束事务。由事务服务的支持来保证create()方法中的数据库操作的可靠性和一致性。

事务属性有以下几种：Required, RequiresNew, Supports, Mandatory, NotSupported, Never。对每一种事务属性，应用服务器会做出相应的反应策略来满足用户的需求，参见下图：

TRANSACTION ATTRIBUTE	CLIENT'S TRANSACTION	BEAN'S TRANSACTION
Required	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Supports	none	none
	T1	T1
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Never	none	none
	T1	error

从表中可以看到Required属性的含义是如果客户端没有发起一个事务，那么服务器会在调用Bean的方法前启动一个新的事务；如果客户端已经发起了一个事务，那么这个事务会传播到被调用端，Bean的方法就在客户事务上下文中执行。其他事务属性的含义都可以从表中得知。客户根据自己不同的需要可以在相应构件的ejb-jar.xml文件中设置合适的事务属性。在构件描述文件中声明事务服务需求也支持通配符表示构件的所有方法，以及支持对同一方法名，不同参数的方法的事务声明。

14.8.1.2. 编程式

Standalone客户，Web客户和Enterprise Bean中的Session Bean及MessageDriven Bean还可以使用编程式访问事务服务。客户通过字符串”java:comp/UserTransaction”利用名字服务得到javax.transaction.!UserTransaction这个JTA接口的实现，然后调用!UserTransaction的begin()方法启动一个事务，commit()方法提交一个事务，rollback()方法回滚一个事务以及其他一些方法来访问和控制一个事务。Enterprise Bean中的访问流程如下所示：

```
Context ctx = new InitialContext();

UserTransaction utx= (UserTransaction)ctx.lookup("java:comp/UserTransaction");

utx.begin();

Connection conn = getConnection();

PreparedStatement ps = conn.prepareStatement("insert into accounts values(?,?,?)

ps.setString(1,"123");

ps.setString(2,"456");

ps.setDouble(3,789);

ps.executeUpdate();

utx.commit();
```

对于Session Bean和MessageDriven Bean除了通过名字服务得到UserTransaction接口外，还可以通过EJB Context得到这个接口：

```
protected SessionContext ctx;

……

UserTransaction utx = this.ctx.getUserTransaction();

……

utx.begin();

//do something

utx.commit();
```

14.8.2. 使用安全

14.8.2.1. 安全机制

EJB规范定义了完整的安全机制来控制用户对每一个EJB方法的访问。EJB规范不赞成EJB开发者将安全控制硬编码到EJB程序中，而鼓励遵循规范在部署文件中配置安全访问权限，这样可以使EJB能够灵活地部署到各种操作环境中。

为了简化部署者(Deployer)的工作，应用程序装配者(Application Assembler)可以定义安全角色(security roles)。一个安全角色是为假定的用户设置的一组访问权限，应用程序装配者可以在安全角色中详细定义对每 一个EJB方法的访问权限。部署者在部署EJB时，只需要将用户或组简单地对应到安全角色上。

14.8.2.2. 定义安全角色

使用security-role元素来定义安全角色

用role-name元素给角色命名

使用description描述角色(可选)

定义的安全角色在ejb-jar整个文件范围有效，即在同一个ejb-jar文件的所有的ejb都可使用这些安全角色。

14.8.2.3. 方法授权许可

定义安全角色后，可以通过method-permission元素指定相应角色所能进行的方法调用。

method-permission元素包括下列子元素：

- role-name：角色名称
- method：授权调用的方法
- ejb-name：方法所在的构件
- method-name：方法名称

14.8.2.4. 定义用户

第三节定义的角色需要委派给服务器的用户，因此，首先必须在服务器中创建用户。注意和大多数应用服务器不同，PKUAS中没有组的概念。这主要源于这样的考虑：安全角色本身就是一个抽象的用户组，一个安全角色可以委派给多个用户。对于大多数应用而言，通过将角色委派给用户已经可以满足需求。

PKUAS通过JAAS机制实现用户认证，目前服务器提供的缺省认证模块为pku.as.securety.login.UsersRolesLoginModule。该认证模块对用户名、密码进行认证。

PKUAS提供了一个命令行节目创建用户。通过运行PKUAS安装目录下bin目录下面的useradmin.bat（windows平台）或useradmin.sh（linux/unix/Solaris平台），实现用户管理功能。

14.8.2.5. 将角色委派给用户

创建了角色、用户，还必须将不同的角色委派给相应的用户，这必须在（pkuas.xml） application.xml中添加项目security-realm进行“角色委派”：

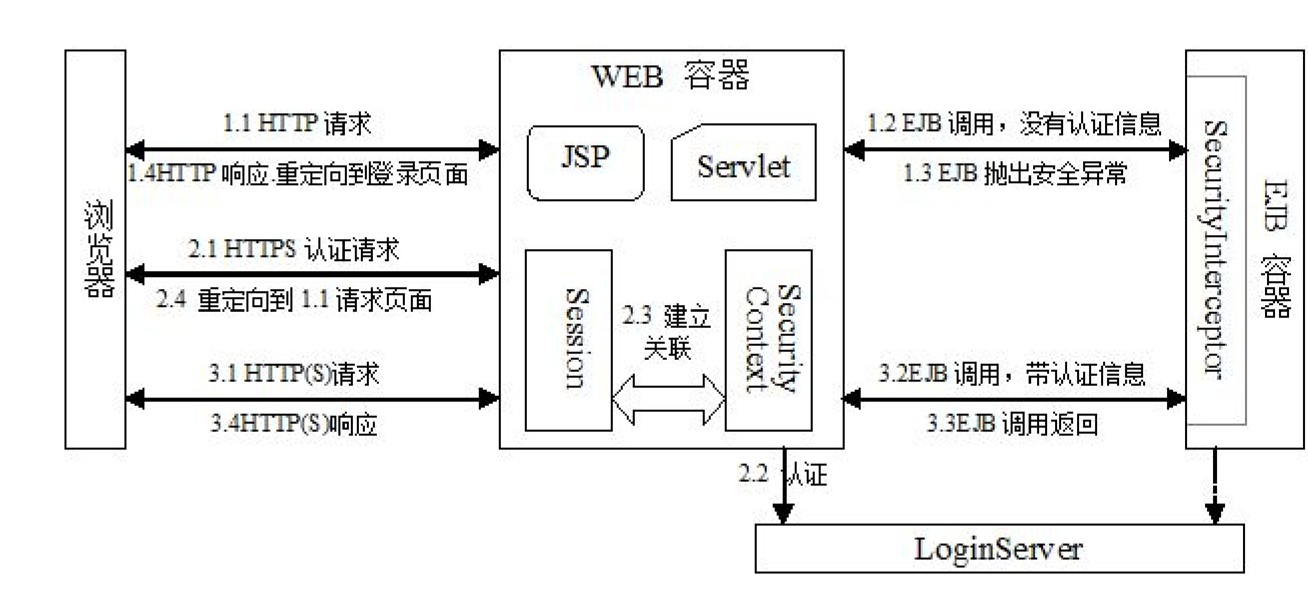
security-realm包括下列子元素：

- realm-name：安全域名称
- mapping：将哪个角色委托给哪个用户

14.8.2.6. 使用PKUAS的“单次登录（Single signon）”

14.8.2.6.1. pkuas中的WEB访问流程及“单次登录”

PKUAS对Servlet规范中定义的标准安全性进行了扩展。如果应用开发者没有进行如图3所示的页面保护配置，则意味着安全检查延迟到对EJB构件访问时进行。WEB容器会捕获对EJB构件访问时引发的安全异常：如果安全异常的原因是用户没有登录，则将请求重定向到登录页面。WEB容器获得用户的身份凭证信息然后，到认证服务器认证，并建立本次会话的安全上下文。WEB容器以后发出的EJB调用，都携带该安全上下文。



上图描述了用户通过WEB方式对PKUAS中受保护的EJB构件进行访问的一次流程：

用户对某JSP页面发出HTTP请求（1.1），WEB容器接收到该HTTP请求，并进行分析；

如果该页面没有被保护，JSP或者Servlet对象将调用后台的EJB构件，进行EJB调用（1.2）；如果该页面被保护，则转到4）；

由于EJB调用（1.2）访问的EJB构件受保护，而此时用户没有登录，因此EJB调用请求（1.2）中没有安全认证信息，EJB容器负责安全检查的SecurityInterceptor将引发安全异常。该异常被WEB容器捕获后，转4）；

WEB容器将HTTP请求（1.1）重定向到登录页面，要求用户输入身份认证信息。登录页面的访问通过安全传输协议HTTPS进行；

用户输入用户名/密码，WEB容器将到认证服务器LoginServer进行认证（2.2），得到认证服务器返回的认证令牌后，建立安全上下文；并建立HTTP会话和该安全上下文的关联（2.3）；最后重定向到原来的HTTP请求（1.1）指向的页面；

用户开始正常的HTTP(S)请求（3.1），当调用EJB构件时，根据本次会话信息，得到安全上下文，随后进行的调用请求（3.2）中，将携带相关认证信息；

EJB容器进行安全检查。通过后，返回本次调用结果（3.3）；

浏览器得到响应（3.4），本次调用结束。

除非用户关闭本次会话，否则该浏览器随后发出的对当前应用的HTTP调用，都不必让用户输入安全认证信息，这就是所谓的“单次登录”（Single Sign-on）。WEB容器负责对一次会话进行管理。

从上面的WEB接入流程可以看出，应用服务器在WEB容器和EJB容器内分别进行了安全检查。在PKUAS中，浏览器和WEB服务器之间的传输安全由集成的Tomcat WEB服务器负责，WEB容器和EJB容器使用统一的安全认证设施。

14.8.2.6.2. 使用“单次登录”

在PKUAS使用单次登录，需要在WEB端的部署描述信息中加入有关安全的信息，以及提供用于用户登录和登录失败后显示的两个页面。

在（pkuas.xml）application.xml中描述安全信息入下所示：

```
<web-module>

        ……

        < auth-method>form</auth-method>

<form-longin-config>

                <login-page>login.jsp</login-page>

                <error-page>error.jsp</error-page>

        </form-login-config>

</web-module>
```

form-longin-config元素中声明用于登录的JSP页面和登录失败后返回的错误页面。

一个简单的登录页面如下所示：

```
<html>

<head>

<title>Login Page for Eshop</title>

<body bgcolor="white">

<form method="POST" action='<%= response.encodeURL("j_security_check") %>' >

  <table border="0" cellspacing="5">

    <tr>

      <th align="right">Username:</th>

      <td align="left"><input type="text" name="j_username"></td>

    </tr>

    <tr>

      <th align="right">Password:</th>

      <td align="left"><input type="password" name="j_password"></td>

    </tr>
```

```
      <td align="right"><input type="submit" value="Log In"></td>

      <td align="left"><input type="reset"></td>

    </tr>

  </table>

</form>

</body>

</html>
```

其中，包含的form的action必须为response.encodeURL("j_security_check")。

附录 A. 附录

本规范参考以下文档:

[1]AAAA

[2]BBBB

[3]BBBBBBB